Developing a Test Suite to aid in Single Event Effect testing of ARM microcontrollers

Victor J Sciocatti

Thesis presented in fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University.

Supervisor: Dr Arno Barnard
Department of Electrical and Electronic Engineering

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2021

Plagiarism Declaration

1. Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.
2. I agree that plagiarism is a punishable offence because it constitutes theft.
3. I also understand that direct translations are plagiarism.
4. Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
5. I declare that the work contained in this dissertation, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for obtaining any qualification.

Abstract

This thesis presents a test suite's design and development that can easily integrate Arm Cortex-M, specifically the STMicroelectronics STM32-range, into the SEE test environment at iThemba Labs with minimal effort from the testers. It can also help local satellite designers verify the resistivity of their designs to Single Event Effects (SEE), even with minimal, or no, SEE test data for their specific Device-Under-Test (DUT). A concept design for such a system is done, and an iterative approach is followed to implement this system. The validity of using JTAG to extract or inject data into the DUTs is investigated. A custom JTAG driver is reverse-engineered and implemented to be able to use JTAG to quickly extract data from a DUT to perform low-level data extraction to pinpoint SEE induced errors, or inject data into a DUT to emulate a SEE test environment and to verify mitigation techniques and system responses before SEE testing. A test station, utilising a Raspberry Pi, is designed to interface with the DUT while inside the SEE test radiation vault and has the option of controlling power to the DUT as well. A monitoring station is also implemented that allows the tester to interface to the test station from a computer over ethernet, enabling safe operation of the test station and DUT from outside the vault. The design is easily changeable to allow for different testing styles or DUTs. Before the final iteration, it was also subjected to a real SEE test at iThemba Labs to verify its effectiveness. A scripting ability was also added, allowing for automated tests that can be useful to increase testers' effectiveness during SEE testing or to use fault injection to determine areas sensitive to errors inside DUT implementations or to test the effectiveness of implemented mitigation techniques.

Uittreksel

Hierdie tesis detaileer die ontwerp en implementering van ‘n toets stelsel wat maklik gebruik kan word om Arm Cortex-M, en spesifiek die STMicroelectronics STM32-familie, mikroverwerkers te integreer in die Enkel Gebeurtenis Effek (EGE) toetsomgewing by iThemba LABS, met minimale moeite en veranderinge deur toetsers. Dit kan ook gebruik word deur plaaslike satelliet ontwerpers om die weerstand van hulle ontwerpe teen EGE meer effektief te verifieer, selfs met geen of minimale beskikbare EGE toetsdata vir die spesifieke Toestel-Onder-Toets (TOT). ’n Konsep ontwerp vir so ’n stesel word gedoen, en ’n iteratiewe ontwerp implementering word gevolg om die stelsel op die been te kry. Die geldigheid van JTAG om data te onttrek of in te voeg na die interne componente van die TOT word ondersoek, en ’n JTAG drywer word ontwerp en gebruik om vinnig data te lees vanaf ’n TOT vir lae-vlak fout analise om EGE foute presies op te spoor, of om data in te ent om die effekte van die EGE toets omgewing te emuleer en om toegepaste mitigasie tegnieke se effektiwiteit te bepaal en die stelsel reaksies te karakteriseer voor EGE toetsing plaasvind. ’n Toetsstasie, wat gebruik maak van ’n Raspberry Pi, word ontwerp wat kan kommunikeer met die TOT terwyl beide in die EGE toets radiasie kluis is. Die toetsstasie kan ook die krag toevoer na die TOT beheer. ’n Monitor-stasie word ook geimplementeer wat die toetser toelaat om deur middel van ’n rekenaar via ethernet te koppel aan die toetsstasie vir veilige beheer oor die toetsstasie en die TOT vanaf buite die kluis. Die maklik-veranderbare ontwerp maak voorsiening vir verskillende toets-style of TOTs, en voor die finale iterasie word die stelsel ook blootgestel aan ’n EGE toets by iThemba LABS om die stelsel effektiwiteit te bepaal. ’n Skrip-vermoë was ook bygevoeg wat voorsiening maak vir outomatiese toetsing, wat waarde kan byvoeg vir toetsers gedurende EGE toetsing, of deur fout inenting te gebruik om sensitiewe areas vir EGE effekte binne-in die TOT te bepaal. Dit kan ook help om te verifieer of geimplementeerde mitigasie tegnieke werk.

Acknowledgements

I want to thank my supervisor, Dr A. Barnard, for his insight, assistance, guidance and wisdom. I would also like to thank my family for their unwavering encouragement, especially during the Covid-19 lockdown.

Contents

List of Figures

List of Tables

Acronyms

ADC	Analog-to-Digital Converter, 23, 24, 38, 39, 47, 48
CAN	Controller Aided Network, 19, 21, 22, 39, 40, 47, 48
COTS	Commercial-Off-The-Shelf, 3 - 5, 8, 12, 17, 18, 24, 29 - 32, 89
CPU	Central Processing Unit, 28, 111
CRC	Cyclic Redundancy Check, 20, 22, 38, 47, 70, 85
DUT	Device-Under-Test, 3, 6, 9, 12, 15 - 17, 27, 29 - 38, 40 - 44, 46 - 49, 52 - 63, 65 - 67, 69 - 75, 78 - 80, 82 - 91, 109 – 111
FLNR	Flerov Laboratory of Nuclear Reactions, 12
FPU	Floating-Point Unit, 19, 32
GUI	Graphical User Interface, 43, 45, 47, 49, 51, 52, 67, 68, 71, 72, 74, 76 - 78, 81, 88, 89
IDE	Integrated Development Environment, 19, 37, 51, 52, 53
I2C	Inter-Integrated Circuit, 23, 40, 47, 48
IC	Integrated Circuit, 25, 28
iTL	iThemba LABS, 4 - 7, 9, 13 - 16, 18, 29, 32, 44, 45, 57, 58, 69, 74, 76, 77, 79, 81, 87, 88, 90, 92
JINR	Joint Institute of Nuclear Research, 12, 13
LEO	Low Earth Orbit, 2, 4, 7, 8, 11 - 13, 18, 36, 40, 41
LET	Linear Energy Transfer, 9, 12 - 13, 30, 69, 70, 83, 86
LFI	Laser Fault Injection, 28, 29
MCU	Microcontroller unit, 4, 5, 7, 8, 11, 15, 17 - 20, 22 - 40, 42, 43, 45 - 56, 60, 61, 65, 67, 69, 72, 82, 83, 85, 87 - 89, 91
MOSFET	Metal-Oxide-Semiconductor-Field-Effect-Transistor, 10, 62
PC	Personal Computer, 29, 46, 47, 48, 49, 51, 52, 54, 55, 56, 57, 67, 77, 88, 109
PCB	Printed Circuit Board, 25
POC	Proof of Concept, 5, 6, 35, 40, 43, 45, 46, 48 - 50, 52 - 54, 56 -58, 60, 61, 67, 74, 87, 88
SAINTS	South African Institute of Nuclear Technology & Sciences, 13
SANSA	South African National Space Agency, 2
SEE	Single Event Effect, iii, 4 - 9, 11 - 20, 22, 26, 27, 29, 30, 32 - 34, 36, 40 - 42, 44 - 47, 53, 56, 57, 61, 62, 69, 70, 72, 74, 75, 79, 81 - 85, 87 – 92

SEFI	Single-Event Functional Interrupt, 11, 34, 40, 41, 46, 48
SEL	Single-Event Latchup, 10, 13, 14, 30, 43, 44, 57, 62, 82, 89
SET	Single-Event Transient, 10, 40, 69, 70, 82, 83
SEU	Single-Event Upset, 10, 11, 14, 20, 29 - 31, 33, 34, 36 - 41, 44, 51, 53, 67, 69, 70, 73, 75, 82 - 85, 87 - 90
SPI	Serial Peripheral Interface, 22, 23, 40, 47, 48
SRAM	Static Random-Access Memory, 3, 14, 17, 28, 32, 53, 61, 73, 82 – 86
SSH	Secure Socket Host, 58, 64, 67, 68, 71, 74 - 79, 81, 88
STM	STMicroelectronics, 18, 19, 26, 37, 53
SWD	Serial Wire Debug, 27
TAP	Test Access Port, 25, 26, 49, 54
TI	Texas Instruments, 30
TID	Total Ionising Dose, 7 - 9, 11, 14, 30, 38, 39, 62, 91
TMR	Triple Modular Redundancy, 85, 86
UART	Universal Asynchronous Receiver-Transmitter, 20 - 22, 29, 32, 35 - 49, 51, 52, 57, 62, 79, 85, 86, 88, 90, 108, 109, 111
USB	Universal Serial Bus, 9, 26, 30, 37, 44, 46, 57 - 59, 62, 63, 91

Chapter 1
Introduction

As electronic devices shrink in both size and cost, and the drive in developing countries, like South Africa, to showcase their abilities to reach for the stars grow, it becomes simpler and more affordable for universities, national research organisations, and even private companies to expand into satellite and space technologies.

This chapter introduces a brief look at the South African history in satellite development and the need for Commercial-Off-The-Shelf components to be used in these designs. Then a overview of ionising radiation testing is given, followed by the need for more effective ionising radiation testing in the South African context. Finally, the problem statements, research questions and document layout is discussed.

1.1 The Growth of South African micro-and nanosatellites

In 1980, South Africa started a military-driven space program, planning both a launcher and an earth observation satellite. Despite producing excellent space test facilities, the military program was halted in 1994 due to a political shift before building any flight-ready satellites. The development then shifted to the local universities, with Stellenbosch University launching Africa's first orbiting microsatellite, SUNSAT-1, in February 1999. SUNSAT-1 operated from 1999 to 2001 when contact was lost [1].

In 2009 the second locally produced microsatellite, Sumbandila-Sat, developed and built by SunSpace, was launched. SunSpace was a Stellenbosch University spin-off company, and their microsatellite delivered more than 1200 images before a power switch failure caused a stability loss in 2011.

In 2009 the Cape Peninsula University of Technology also started a satellite engineering program and later collaborated with the South African National Space Agency (SANSA) Space Science Directorate and Stellenbosch University to develop ZACUBE-1, the first South African nanosatellite, which was launched in 2013 [2].

In 2010 the South African National Space Agency was established to coordinate and implement the national space program. Since then, the emphasis was placed on growing the South African presence in space development. The local government awarded SANSA a significant R4.47 billion in 2020 to develop a Space Infrastructure Hub as part of their Sustainable Infrastructure Development Symposium [3].

1.2 The need for Commercial-Off-The-Shelf components

Even though there is active growth in the South African space sector, funding is still limited, and only some 180-200 people are currently involved in engineering space-related activities [4]. As a result, there is a limited number of resources to develop every needed component from the ground up.

However, modern Commercial-Off-The-Shelf (COTS) components are readily available. They can satisfy most satellite design needs, allowing designers to retain the same capabilities while significantly reducing cost and development times.

Since most COTS components have a design focus on being as space-efficient and affordable to produce in bulk as possible, especially for cell phone and automobile use, this allows a significant weight reduction of satellites and faster development times.

Using COTS components may not be a perfect solution. COTS components cannot necessarily handle the extreme temperature swings found in space and are usually not tested for radiation sensitivity. Yet, this is much less of a concern for micro-and nanosatellites as they typically orbit in the Low Earth Orbit (LEO), which is much less harsh than outside the magnetosphere shielding.

Much of the satellite development in South Africa focuses on researching certain technology pieces or showcasing the countries' growth in potential. Mitigation techniques could be applied to harden these satellites' overall design as much as possible while still using COTS components, as seen on Sumbandila-Sat [1].

To implement these mitigation techniques, the components used will need to be tested for radiation sensitivity to identify the areas that require mitigation. Unfortunately, ionising radiation testing is hard to implement as available testing facilities are scarce.

1.3 Ionising radiation testing

Testing electronics for radiation sensitivity can be challenging, especially in developing countries where local, capable facilities are seldom available. Further, depending on the technology used inside a specific Device-Under-Test (DUT), a slight configuration change can have a radical effect on the entire system's radiation sensitivity.

It is almost impossible to define a global model that fits all use cases to predict radiation sensitivity. There is a need for each system to be thoroughly tested before sent into orbit. Typically, there are two main branches of ionising radiation testing.

Firstly, an individual component analysis is when only individual components like single SRAM cells are tested and give a fundamental understanding of how they react in a hostile environment. This type of analysis typically does not predict a response if the tested component is used in conjunction with any other components. The more complex a system, the more effects will be noticeable that the foundational component analysis will not directly characterise.

Secondly, overall system analysis tests the entire system as a unit instead of characterising the individual components in isolation. Usually, this gives a reasonable estimation of a system's fault rate but makes it extremely hard to detect in which area of the system specific errors occurred.

A developer can create a mitigation strategy by using these two approaches in tandem by determining when an error in a system occurs and a thorough knowledge of the foundational elements' response.

This approach does become extremely difficult when using COTS components, as manufacturers might not necessarily divulge how a part's manufacturing is implemented on a foundational level. For complex systems like microprocessors with thousands of sub-components, it becomes virtually impossible. Even different compile methods can lead to vastly different radiation sensitivity results as various sections of the processor are used or disabled. As a result, designers seeking insight into the radiation sensitivity of systems constructed from COTS components typically only have the option of an overall system fault rate.

A further complication for ionising radiation testing is that different sub-atomic interactions can lead to radiation-induced faults. The entire space radiation spectrum's full testing would require several tests and would need various separate testing facilities.

1.4 The need for more effective testing methodologies

An ideal development plan to achieve good radiation mitigation results would be to design a system and perform the full spectrum of radiation sensitivity tests to identify sensitive areas. The developer could then apply mitigation, and the system could be retested to verify the effectiveness of the modifications. This process can then be repeated until a satisfactory level of mitigation is achieved.

In the South African context, with limited facilities, such an approach is unpractical. Instead, designers tend to mitigate what they think is significant and avoid full spectrum ionising radiation testing process, due to the lack of both adequate facilities and testing knowledge. This approach leads to problems once the system is exposed to the harsher environment of LEO and can even be lethal to the system should the data corruption occur in a vital part of the system, such as a bootloader.

There is a need for practical, accessible ionising radiation testing that, at a minimum, can give an estimation of a systems radiation response, including where the faults occur, preferably without having to go through the entire testing process repeatedly.

1.5 Problem statements and research questions

To date, no South African-built nanosatellite has been completely tested for durability against ionising radiation found in the harsh environment of LEO and outer space. However, most of these satellites have experienced Single Event Effect (SEE) induced failures during their missions [15].

The shortage of SEE testing can be attributed to a lack of SEE test facilities and a lack of viable SEE testing methodologies available to South African satellite designers. Some tests have been done ad hoc at iThemba LABS (iTL), in Cape Town, South Africa. Until recently, there were no dedicated test standards or methodologies that could be followed to get consistent, repeatable results.

However, SEE testing at iTL has very recently become a viable option due to the work presented in [15], making SEE testing of COTS components, especially critical system components like microcontroller units (MCU), more accessible to local research and

development groups. These groups cannot afford full radiation-hardened systems and rely on these COTS components in their designs.

To aid South African researchers and satellite designers to be able to perform SEE tests on some of the most critical core components, the MCUs, the following three main research questions are asked:

1. Which COTS MCUs should the local SEE testing be focused on?
2. How can effective, low-level microprocessor SEE testing be integrated into the iTL environment?
3. How can tools like fault injection be used to help testers prepare for SEE testing and verify their mitigation techniques?

The rest of this document will describe the research conducted and the solution developed to answer these questions. The solution required multiple iterations on developing a SEE data extraction and fault injection test suite to aid satellite designers in better preparing for MCU SEE testing and simplifying the testing interface during SEE testing to optimize the scarce beamtime.

1.6 Document Layout

The layout of this document is summarized as follows:

- Chapter 1: This chapter presents the introduction and motivation for this research and the research questions.
- Chapter 2: The literature study presents and discusses relevant literature in this chapter, such as the radiation effects in semiconductors and why COTS processors need to be tested before being used in South African satellites. An argument for which COTS MCUs the testing should focus on is presented, followed by a quick analysis of SEE testing methods of similar devices at other institutions.
- Chapter 3: This chapter explores the concept design of a data extraction and fault injection test suite to aid designers in testing MCUs. The design requirements and drivers are stated, along with the desired functionality to reach these requirements. Possible implementations of the various parts of the system are then discussed.
- Chapter 4: This chapter presents the implementation of a Proof of Concept (POC) design that showcases the foundation the system will be built on. Focus is placed on data extraction and injection methods, as that is the core component of this system, primarily through access provided through a technology called JTAG.

- Chapter 5: This chapter improves the POC to a working prototype, which gains the needed functionality to interface between the tester and the DUT and is then tested in a real SEE test at iTL to verify its effectiveness
- Chapter 6: This chapter further improves the prototype of the previous chapter, making the system more useable for a variety of situations, and fixing the problems that arose during the testing of the prototype.
- Chapter 7: This chapter speculates how data injection can be helpful to satellite designers.
- Chapter 8: This thesis concludes with some thoughts about implementing the latest iteration of the system and highlights some areas of interest that can be the basis of future research and improvement on this project.

Chapter 2
Literature Review

Before any solutions can be sought to aid the SEE testing of microcontrollers, it is critical to first inspect the testing environment, look at research that has already been done, and investigate the available technologies that are available to be used.

This chapter takes a more in-depth look at the technologies surrounding ionising radiation testing, specifically with microcontroller units (MCU) in mind. The focus will be on the specific mechanics, types of testing, testing facilities available, South African-based facilities, and investigate proton beam testing approaches at iThemba LABS *(iTL). Then MCUs are discussed along with some technologies needed to create this project, and examples of other research endeavours with MCU SEE testing.*

2.1 Brief introduction to the effects of ionising radiation in semiconductors

Since the transistor's introduction in 1948, technological advances have led to incredibly intricate architectures, smaller dimensions, higher density integrations, lower voltage supplies, and much higher operating frequencies [5, 6]. These improvements have led to digital systems that can execute complex objectives while using less space and power than ever before and are more reliably and economically produced.

For most terrestrial applications, this is highly beneficial. For extra-terrestrial use, this also led to reduced transistor reliability in the harsher Low Earth Orbit (LEO) and space environments by being more susceptible to radiation interference-induced faults caused by energised particles, protons, and heavy ions. These charged particles or secondary particles, such as alpha particles from neutron collisions inside the device [7], interact with the internal components and can be credited to tightened noise margins and reduced threshold voltages and node capacitances [8, 9]. These interactions can lead to onboard radiation damage in satellites if left unchecked. Onboard radiation damage can be grouped into Total Ionising Dose (TID) and Single-Event Effects (SEE).

TID is a cumulative effect of long-term device degradation when exposed to ionising radiation. It is not a primary focus of this research, but is discussed in the next section as, for South African ionising radiation testing context, testing for both TID and SEE will typically be scheduled together due to logistical reasons. It is briefly investigated because, should some elements of testing be interchangeable, it might simplify the preparation for the testers.

In contrast, SEE occurs when an individual ionising particle interacts with a device and causes an effect [10], either through direct or indirect ionisation. Direct ionisation is where a charged particle directly deposits enough energy to cause an effect by freeing electron-hole pairs along its travel path through the semiconductor material. Indirect ionisation is where the fault occurs by triggering a secondary nuclear reaction in an inelastic collision with a target nucleus as it enters the semiconductor lattice [11].

2.2 Total Ionising Dose testing

TID testing is available locally to South Africans and occurs more frequently than SEE testing. Due to the proximity of the TID and SEE testing facilities, tests can be scheduled together for logistical reasons, especially for testers that need to travel to reach these facilities in Cape Town. It will be a bonus if the design of this project can accommodate some aspects of TID testing to try to accommodate testers to only test MCUs with one system, aiding their preparation time. For this reason, a brief overview of TID testing, as used in South Africa specifically, is made. Complete integration of the system designed in this project into a TID testing environment is outside the scope of this project

TID is the cumulative damage of a semiconductor lattice caused by ionising radiation over time. In a space radiation environment, this damage is caused by high-energy particles [21]. For terrestrial testing, irradiation with high-energy photons from a Co^{60} gamma-ray source can be used to simulate the harsh space environment [22].

TID results in constant damage to the system through trapped charge produced by the ionising radiation, caught in the semiconductors' dielectric material. Eventually, this damage can cause parametric and functional failures in microcircuits through changes such as shifts in the threshold voltage, gain reductions, or increased leakage currents [23]. Typical dose rates for LEO satellite systems developed in South Africa should be around 2kRad per year. Thus, for a lifespan of 5 years, satellite systems should be able to withstand a total dosage of more than 10kRad [24].

As described by [24], Stellenbosch University has been involved with TID testing of COTS components since 2000 and tested more than 30 devices between then and 2007. The setup

described could deliver between 2.42kRad/h, 1.5m from the source, to 15.1kRad/h at a distance of 0.6m. The possible large diameter from the source allows for irradiation of several DUTs simultaneously. The dose rate can be determined accurately due to movement along the increased radius between the DUTs and the source, compared to smaller setups, such as described by [25] at the University of Saskatchewan that has a chamber size of 0.152m (diameter) by 0.206m (height) and a dose rate of 16.2kRad/h (4.6 Rad/s).

For the physical testing at [24], the cylindrical Co^{60} source can be pneumatically lifted from or retracted into an underground shielded housing. When raised, the entire room is exposed to the source, and the researcher chooses the most desirable distance from the source for the TID experiments. Communication with the DUTs need to be established from outside the testing area as the exposure is hazardous to humans. The measurements were taken by a university-developed data acquisition system that was placed inside the radiation chamber. A laptop remotely controlled it from outside the radiation chamber through either an RS232 or USB connection.

The parameters tested to verify the TID effects on the DUTs vary depending on the devices tested. Processor functionality tests were done for various peripheral and core components, including supply currents, I/O write-read, memory tests, and peripheral loop-backs.

2.3 Single-Event Effect Testing Overview

2.3.1 Important definitions

Two essential terms need to be understood to discuss and model the effects of ionising radiation on specific electronic circuits. Though not used extensively in this document they are critical in understanding the work done in [15], which is the basis for SEE test setups at iTL.

These terms are briefly explained as follow:

- A cross-section is a measure of the probability that a specific process will occur when some radiant excitation intersects a local phenomenon [18], or simply how likely is it particles will interact with each other in a given way [19].
- Linear Energy Transfer (LET) is the average (radiation) energy deposited per unit path length along the track of an ionising particle [20].

2.3.2 Types of SEE

SEE can be divided into various groups. Some effects are non-destructive and only have temporary effects. These effects are called soft errors [13]. Likewise, some effects can be destructive to the system, and these are called hard errors. [13] Some of these destructive effects can have a temporary impact if caught and reset quickly enough, and others are devastating to the system should they occur [12].

The soft errors are Single-Event Upsets (SEU), Multiple-Bit Upsets, and Single-Event Transients (SET). While the hard errors are Single-Event Latchups (SEL), Single-Event Snapback, Single-Event Burnout, and Single-Event Gate Rupture. These effects can be summarised as follows:

- Single-Event Upset is a temporary change of state in a memory or control bit [14] of a storage element due to the effect of a single charged particle strike. The particle strike incurs no damage, and the state can be corrected by overwriting the corrupted bit to its original value [16]. An SEU can be directly induced through deposited charge from the charged particle in a critical transistor of a storage element circuit, as shown in Figure 2.1. It can also occur due to a SET captured by a storage element [15].
- Multiple-Bit Upset is a temporary change of state in multiple memory or control bits of a storage element due to the effects of a single charged particle strike. It operates on the same mechanism as SEU.
- Single-Event Transients are voltage glitches in circuits caused by the disposition of charge near a transistor's charge-sensitive area. This charge is amplified by the transistor and incorrectly stored further along the circuit by a memory element [15][16].
- Single-Event Latchup occurs when a current path is established by forming a thyristor structure in a transistor during charge disposition [16]. SEL can be non-destructive if detected and mitigated quickly enough by removing power to the transistor before the current leak generates enough heat to damage the element or be destructive otherwise.
- Single-Event Snapback occurs at the drain junction of an N-channel power Metal-Oxide-Semiconductor-Field-Effect-Transistor (MOSFET) and is similar to SEL [14].
- Single-Event Burnout occurs when the drain-source voltage exceeds the breakdown voltage in a power MOSFET due to the substrate close to the transistor's source becoming forward biased [15].
- Single-Event Gate Rupture happens when deposited charge near high-intensity electric fields lead to a current path destroying the gate or a dielectric layer of high power MOSFET devices [15].

These effects can also lead to a Single-Event Functional Interrupt (SEFI), which is not an effect in itself, but rather the result when any SEE triggers a loss in system functionality, leading to the system functioning incorrectly [15].

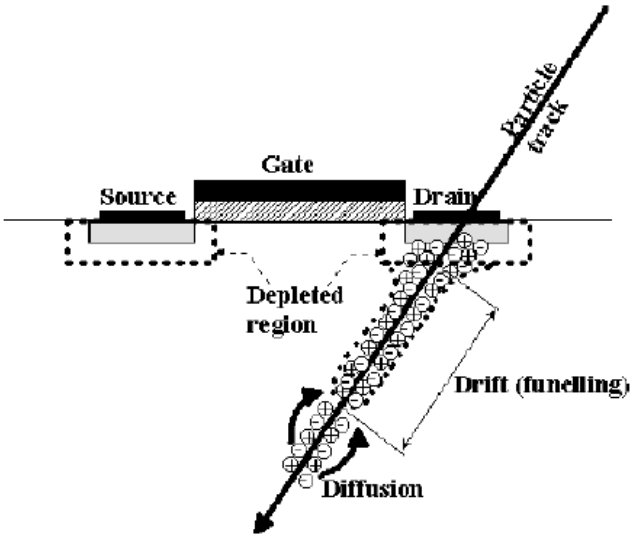


Figure 2-1 A representation of an SEU strike [17]

2.3.3 SEE testing variations

SEE testing is usually done through the use of particle accelerators. A particle accelerator propels and accelerates ions and charged particles such as protons to speeds close to lightspeed. Particle accelerators direct these particles' movement direction by using magnetic fields, and acceleration is possible through changes in electric fields the particle is moving through [26].

The particles can be energised due to their inertia and can be guided towards and then smashed into targets to investigate the collision effects. For semiconductors, as found inside MCUs, the effects mentioned in Chapter 2.1.2 can occur when the charged particles bombard the semiconductor. Particles of different sizes and energies will cause other SEE effects.

Testers must do several tests to get a semiconductor response model equivalent to LEO exposure, using various types of charged particles. Without these tests, the testers have no way to know which components are vulnerable to SEE. As such, they will also not know on which subcomponents to apply SEE mitigation techniques.

There are several types of charged particle beams. Each type has advantages and disadvantages when used for SEE testing.

2.3.3.1 Heavy Ion Beams

Heavy-ion beams are obtained in particle accelerators by accelerating charged nuclei heavier than protons to increase the ion energy to ionising radiation levels. Carbon-ion beams are used in radiotherapy, mostly in Europe and Asia [30] for cancer therapy by accelerating carbon ions to 70% of the speed of light and irradiating the target cancer cells while minimising the dose to the adjacent critical organs [27].

For replicating the SEE sensitivity of electronics in an extra-terrestrial environment, testers can use heavy-ion beams to study the mechanisms contributing to SEEs and estimate in-orbit error rates. SEEs induced from heavy ions typically occur due to the release of electron-hole pairs along the path of energetic charged particles [28].

The direction of heavy ions can be directed using magnetic fields as they are ionised and not charge-neutral. Thus, heavy-ion beams with required energies can be delivered by large accelerators located mainly at basic physics research laboratories. The ions are navigated through the accelerator and fired directly at the target DUT. Accelerators are restricted by the maximum speed of ion acceleration. Changing the ion changes the energies that can be reached since the mass of the ion changes. It is essential to change the nucleon energy and, subsequently, the LET to enable different heavy ions' penetration levels. The variation in penetration depth ensures the DUT gets proper exposure. Since the energy is known for each type of heavy-ion, an accurate cross-section can be calculated.

From [29] the Flerov Laboratory of Nuclear Reactions (FLNR) Joint Institue of Nuclear Research (JINR) cyclotron can deliver an ion energy range of 3-60 MeV/nucleon and an LET range of 4-5-100 MeV/(mg/cm^2) for SEE testing, and can accelerate $^{16}O^{2+}$, $^{22}Ne^{3+}$, $^{40}Ar^{5+}$, $^{40}Ar^{4+}$, $^{36}Fe^{7+}$, $^{84}Kr^{9+}$, $^{84}Kr^{11+}$, $^{132}Xe^{11+}$, $^{132}Xe^{18+}$, $^{209}Be^{23+}$ and $^{209}Be^{19+}$. The facility has all the needed infrastructure for SEE testing but typically only switches between different heavy ions once per week. As a result, for an accurate cross-section derived from multiple tests with multiple ions, testing can be time-intensive.

Because of their size relative to other nuclei, heavy-ion testing typically happens under vacuum to avoid air molecule collisions. Heavy ions also do not penetrate deeply into dense, high-Z materials, and as such, COTS components need to be de-lidded. De-lidding can damage the underlying circuitry, and it can be a complicated process to achieve reliable results, possibly affecting SEE testing results. FLNR does have de-lidding facilities.

A research relationship already exists between iTL and the JINR. Several research and growth programs, such as the South African Institute of Nuclear Technology & Sciences (SAINTS), are being organised annually [37].

2.3.3.2 Proton Beams

Conceptually, protons can be thought of as ionised Hydrogen, as stripping Hydrogen of its electron leaves only a proton. This makes it easy to think of proton beams as heavy-ion beams, but with Hydrogen ions instead of heavy ions. These beams are the primary focus of many high-energy cyclotrons [31], and different accelerators can produce proton beams with a wide range of nucleon energies.

Like heavy-ions, proton radiotherapy is used in the treatment of cancers and tumours. While worse at breaking down the tumours, it does do less damage to surrounding healthy tissue. Proton beams are preferred in countries like the United States of America, partly due to different funding procedures and the more complex physical and biological features of heavy ions [32], making it more complex to implement.

For SEE testing while using proton beams, beam energies higher than 180MeV are recommended to trigger SEL [33]. Further, the LET of protons is typically too small to trigger SEE by direct ionisation. Instead, the protons cause secondary nuclear reactions that cause SEE, making it impossible to know the secondary particles' LET. It could be much higher than that of the incident particle. [33]. As a result, it is hard to determine an absolute cross-section, but it is possible to determine cross-sections per proton energies, which can help determine the areas of a system that are sensitive to SEEs and needs extra mitigation.

2.3.3.3 Neutron Beams

Neutrons are the predominant SEE cause between the operating altitudes for commercial aircraft and sea level [34]. Neutrons are charge-neutral and cannot be guided and accelerated in the same way as proton beams or heavy-ion beams. The JESD89 standard recommends two types of neutron beams for SEE testing. Both are generated by bombarding specific targets to get a particular neutron emission response, effectively converting the proton beam into a neutron beam [34].

Spallation neutron sources emit neutrons that cover a large energy spectrum. This spectrum resembles the natural high-altitude neutron spectrum [34]. These sources are created by proton-bombardment of materials like tungsten, lead, and liquid mercury [15].

Monoenergetic neutron sources emit monoenergetic neutrons and can be generated by proton-bombardment of materials like scandium, lithium fluoride, deuterium, and tritium-loaded titanium [15].

2.4 SEE Testing, in a South African context

This section handles SEE testing in South Africa and heavily relies on the work done in [15] as that made reliable SEE testing available to local academics. Almost all information in this section can be found in [15] but is briefly summarised for convenience.

2.4.1 General Testing Standards

Several standards give guidance on testing for ionising radiation-induced effects on electronics. According to [35], there are several critical standards for radiation testing and several other standards that can be used in conjunction with the key standards. Because devices keep changing and testing needs to be continually adapted to adjust to the modern instruments, the standards give a guideline of acceptable testing practices rather than a dedicated protocol. It is up to the tester to align their testing method to these standards, but ESA-ESCC-25100: SEE Test method and Guidelines [35] is a good place to start. The full recommended list is available at [36] at the time of this writing and encompasses guidelines for heavy-ion, proton, and neutron SEE testing, as well as TID testing.

Testing at iTL for this project was done following the testing standards identified and implemented in Chapter 2.6 of [15]. It lists the standards selected when testing specifically at iTL.

2.4.2 Testing in a South African context

South African satellites have been heavily impacted by a lack of proper SEE sensitivity testing. SUNSAT, operational 1999-2001, needed a rewrite of onboard firmware due to SRAM SEUs. SRAM micro SELs forced firmware and operational improvements to keep Sumbandila-Sat working. In-orbit observations of SaudiSat 3, built by SunSpace, also revealed SRAM sensitivities to SEE [15].

To date, no South African-built complete satellite system has undergone SEE testing before launch. The need for SEE testing is increasing, especially with industry demand quickly growing. Some SEE tests have been done on the CubeComputer to verify detected in-orbit anomalies. These tests were only done after the component was in-orbit, and with proper testing before launch the severity and characteristics of anomalies could have been characterised and possibly mitigated. The CubeComputer is built around an

EFM32GG280F1024 MCU from Silicon Labs, based on the 32-bit ARM Cortex-M3 architecture [71].

Until recently, testing for SEEs locally in South Africa, or even on the African continent, was highly unrealistic. The only facility with a sufficient particle accelerator is iTL, which is mostly used for medical purposes and research. However, electronics testing at iTL has only been done on an ad hoc basis [15], and minimal experience and guidelines for electronics testing is available, as it is mostly used for medical purposes and research.

To try and prevent future errors as those found in SUNSAT, Sumbandila-Sat and SaudiSat 3 emphasis will need to be placed on consistent, reliable SEE testing. Ionising radiation testing allows satellite designers to observe MCU vulnerability to SEE before the parts reach LEO, making it easier to fortify vulnerable areas, decide whether the chosen MCUs are fitting for the mission before including them in the design, and to implement and test mitigation strategies to deter data corruption and improve the chance of data correction should SEE occur. The outcome of all these options is that the final implementation of the MCU should be more radiation-tolerant than without ionising radiation testing, and the MCU implementation used in the final design can be expected to have a longer functional life expectancy.

2.4.3 Test setup and procedure at iThemba LABS

Testing at iTL will be done according to [15], under the original author's guidance, and is specific to SEE testing at iTL. It is summarised in this section.

Test slots for SEE at iTL usually have a minimum of 8 hours of beamtime. Personnel needed to conduct tests include the researchers working the tests and all the iTL staff required to operate the facility. Factoring in the power cost to use the beam itself, it can cost R90k/hour, so effective testing is crucial.

The proton beam will be guided through the accelerator system to the test vault under vacuum, where it will exit into the air through a Havar window. Upon exiting the Havar window, the beam spread will be around 3mm to 6mm, which is too narrow to irradiate most MCUs. From this point, the beam shape can be modified by the testers to produce the required setup, as described in the following paragraphs.

Before the test, the following setup is implemented:

A thin lead sheet is placed directly after the Havar window to scatter the beam. The thicker the lead sheet, the wider the angle of scatter. Typically the beam spread needs to be wide

enough that the DUT receives a uniform exposure. A set of collimators is then used to absorb the excess scatter to ensure that only the DUT is exposed in the beam spot. Should the beam energy being delivered be too high for the test requirements, it can be degraded by placing Plexiglass inline between the beam and the DUT. The thicker the Plexiglass, the more significant the energy degradation of the protons.

The distances between the Havar window, the lead sheet, the collimators and DUTs are calculated depending on the distance between the Havar window and DUTs, the size of the available collimators, and the required spot size. Care should be taken to ensure all components are level and aligned correctly.

The DUTs are mounted onto a controllable, movable backboard, and remotely moved in or out of the beam spot. A wooden table is placed near the DUTs, on which support electronics needed to control the communication and power the DUTs are placed. Concrete or lead bricks are packed between these electronics and the beam to give shielding against the scattered protons and secondary radiation.

One of the support electronic devices is an ethernet switch that connects all communication devices inside the vault to the outside via the ethernet cable. The DUT itself is connected to the support and monitoring electronics, which communicates with the testers in a control room. The setup inside the vault is depicted in Figure 2-2.

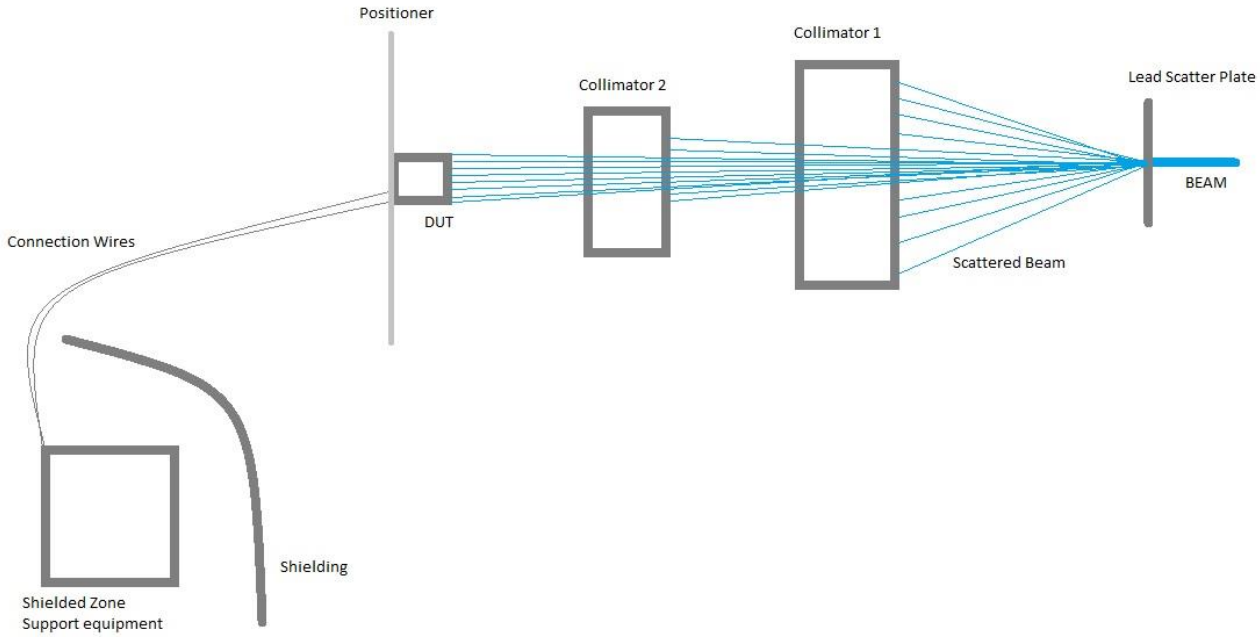


Figure 2-2 Vault setup for SEE testing at iTL

When testing, the following procedure is followed:

The control room gives testers access to raise and lower a Faraday cup inside the beamline, which blocks or allows the beam to exit into the vault. While the beam is being delivered to the vault's inside, all humans must be a safe distance away. No quick changes can be made to the test setup before the radiation levels have lowered significantly, so it is essential that the setup is done correctly and the remote connections are functional.

Before any DUTs are irradiated, the beam control room is signalled to deliver a starting current. The higher the current, the more particles will be directed to the DUT. During the test, the control room can be signalled for any current changes between irradiations.

Once the testers are ready, the Faraday cup is lowered, and a sweep is done using a Beam Loss Monitor to determine the beam profile to verify the spreader and collimators setup is correct. After that, the Faraday cup is raised, and the first DUT is moved into the beam spot. Irradiation of the DUTs is done in runs. A run starts with the testers agreeing on an exposure time and then simultaneously lowering the Faraday cup and beginning the specific DUT test. After the agreed time is reached, the Faraday cup is raised again, and the next run can begin.

After all runs are done and the test slot is completed, the vault is left to dissipate the internal absorbed radiation levels. Once the levels are safe for short human exposure, any necessities inside the vault can quickly be collected. After a few weeks, the radiation levels will be much less severe and safe for longer human exposures, and the setup can be packed away.

2.5 Microcontroller Technologies

Many satellites use a dedicated radiation-hardened MCU as the central controller. They are much more expensive than their COTS equivalents, and often are bigger and use more power. However, COTS MCUs are gaining popularity as secondary controllers due to the cheaper cost and better space and power usage. Being able to replace all secondary onboard MCUs with COTS MCUs that can perform non-critical tasks is vital for a more affordable satellite. As such, this project will focus specifically on COTS MCUs for SEE tests.

Before a test suite can be implemented to test MCU sensitivity to SEEs, it is important to look at what type of MCU is viable for satellite designers to use, what components exist within these MCUs and how they can be tested.

2.5.1 Microcontroller Selection

As mentioned in Chapter 2.4.2, Sunsat, Sumbandila-Sat and SaudiSat 3 all experienced SEE. The relevant errors occurred in internal components, especially SRAM, on the satellite designs' respective MCUs [15]. For this project, the focus will be exclusively on COTS MCUs to aid local designers in properly testing future satellite endeavours. The MCU must be a COTS component as they are currently widely used in the local industry, from the satellites already launched to current component designs from the local companies [71]. Radiation-hardened MCUs are extremely expensive, bigger and require more energy, while COTS MCUs are easy to implement and much more affordable.

2.5.1.1 COTS Microcontroller requirements

Alongside being a COTS component, the following requirements are identified for MCUs targeted for space use:

- MCUs used for satellite micro-and nanosatellites must be small due to space restrictions and highly power-efficient due to power supply restrictions.
- The MCU must also communicate with the rest of the system. It might not be the central onboard controller and only controls individual subsystems and any other peripherals needed to execute its tasks.
- It must operate in an environment readily available to both the academic and industrial communities as SEE testing in South Africa can only be done at iTL, which restricts beamtime for testing to academic research. Industrial developers can then use public academic results to develop their systems.
- It should be affordable and use free or inexpensive programming interfaces to encourage adoption in the industry and suit academic budgets.
- It must be readily available to avoid long lead times and allow rapid development and adjustments, especially for test preparation.
- It must have a built-in interface that can be used to access the boundary registers.
- An active development community will be preferred to aid in quick design and fault-finding.
- It should be interfaced in a way adaptable to other MCUs. Due to the nature of low-level fault finding and injection, some alterations will need to be made by the user for the specific MCU chosen. Ideally, the least number of changes should be required to switch MCUs. As such, the MCU-approach implemented in this project should be easily expandible to similar models of the same brand or even across brands.
- It should preferably also be used or investigated by the industry for LEO satellite use.

2.5.1.2 Selected Microcontroller

Two popular processor families that fit most of the previous section's requirements were identified, namely Arduino and STMicroelectronics (STM) STM32 processors. After an inspection of available MCUs in each range, the two MCU families are compared in Table 2-1.

Table 2-1 Comparison of STM32 and Arduino MCUs for space development applications

Criteria	Winning MCU family
Affordability	Both
Availability	Both
Easily expanded to similar MCUs	Both
Versatile peripherals, especially CAN	STM32
Low-powered	STM32
Ease of use	Arduino
Well-developed IDE	STM32
Supporting Community	Arduino
Industry use	STM32

As shown in Table 2-1, Arduino is initially easier to implement. However, due to this initial simplicity, it quickly becomes much more complex should the designer want access to low-level functionality. The platform was designed to be easy to use to help beginner hobbyists to use the technology, and as a result hides advanced features and avoids low-level access. Low-level access is regularly required for mitigation techniques or complex communication implementations.

Currently, the Arduino community is also bigger, again due to their focus of getting beginners started, but the STM community is quickly growing and is advanced enough to provide plenty of development activity and support should it be needed.

The STM32 MCUs are, for this use case, the better processors to use. They are ARM-based and are similar to ARM-based processors from other brands, thus easily expandible. They have an extensive range of low-powered processors. They are relatively easy to use while still offering programming flexibility and have free established development environments and a growing support community.

Most importantly, a local company has expressed a desire to use the STM32 range in their satellite designs and are interested in the SEE sensitivity of these devices. They are

specifically interested in the STM32L452RE, an ultra-low-power MCU with a FPU Arm Cortex-M4 MCU.

For this project, the STM32L452RE will be used as the chosen test MCU, but care will be taken in the design to allow switching MCUs with relative ease, to avoid limiting the design choices of satellite designers. Switching between various STM32 MCUs should be fluent, and preferably other Arm-based MCUs from other manufacturers as well, as they share very similar designs.

2.5.2 Peripherals

This section will investigate MCU peripherals commonly used in satellites so that simple, suitable methods to verify their functionality can be discussed in Chapter 3.

This project aims to extract data from any memory location in the MCU, including peripherals, to inspect for SEU. However, in most situations, it is not feasible to extract every bit every time an inspection is made as it can take too long to read the entire memory in one go. Thus, it would help to detect which areas are not functional and then only extract those areas to pinpoint where the fault occurred.

These peripherals are not MCU dependent and should apply to all MCUs in most cases. However, specific MCUs might come with an enhanced feature set that allows more specific or effective testing of certain peripherals. An example of this is the built-in Cyclic Redundancy Check (CRC) implementation on the STM32L452RE, which can be used to verify memory contents.

It is the satellite designer's responsibility to choose the best method of implementation for their desired goal. This section does not focus on the specific code used to get their selected MCU operational but rather to inspect how these peripherals operate and give insight into how they could be tested.

For the specific testing approach of peripherals in this project, see Chapter 3.2. The testing method will be focused on exercising the peripheral while the MCU analyses its functionality. This should be handy should the functionality of specific peripherals need to be verified if it performs a critical role.

2.5.2.1 UART

A universal asynchronous receiver-transmitter (UART) is an asynchronous communication device. It has configurable data formats and transmission speeds. It consists of one transmit

and one receive line and is commonly used for communication over computer or peripheral serial ports [38]. Most MCU's contain a minimum of one UART due to its use as a simple debugging interface.

For communication between two UARTs, typically from two different subsystems, the receiver of one is connected to the other's transmitter and vice versa. The transmitter and receiver on both devices both contain a shift register used to convert between serial and parallel forms. The transmitter's data is delivered in parallel from the transmitting device into the shift register and then sent bit by bit, sequentially, to the other device's receiver. Received data is stored in a shift register and restored to a byte, and available to the processor.

If no data is being transmitted, the line is usually held high. When a message starts, the line is pulled low as a start signal followed by the byte, a parity bit, if enabled, and finally, the stop bits. This process repeats for every byte until the entire message is broadcasted. This can be shown in a timing diagram, as in Figure 2-3.

This method of communication is typically slow, given its serial nature and use of slower transmission speed. It can also usually only be used for communication between two devices, and not more. It also has no built-in method for verifying that the entire message, whether transmitted or received, was correctly received.

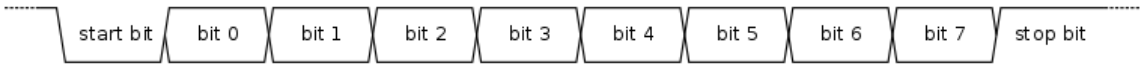


Figure 2-3 UART communication timing diagram example

2.5.2.2 Controller Aided Network

Controller Aided Network (CAN) was initially introduced to reduce wiring in automobiles by being a multi-master, message broadcast system [39]. Unlike UART, CAN does not send messages from only one device to another. It broadcasts short bursts of data, like temperature or RPM, to every device connected to the network, allowing all devices to access the data and process it as is required.

A CAN network topology requires at least two nodes to communicate, and all nodes are connected with a two-wire bus. It can be extremely robust and requires both wires to act in

unison, sending data with one wire pulled to 0V and the other high to indicate a low in the message. This is demonstrated in Figure 2-4.

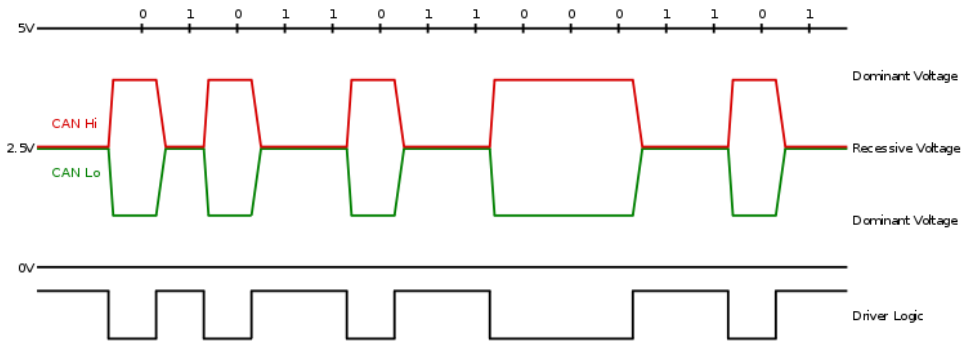


Figure 2-4 CAN Communication timing diagram example [40]

Each node in the network requires a digital signal processor or MCU, a CAN controller, and a CAN transceiver connected to the CAN Bus-Line. CAN messages consist of multiple parts, as shown in Figure 2-5. These parts include 0-8 data bytes, a 16-bit CRC containing the checksum of the message data for error detection, and an acknowledge bit (ACK) that can be used notifying the sender the receiver has detected an error. For a description of the other overhead, see the ISO-11898:2003 Standard.

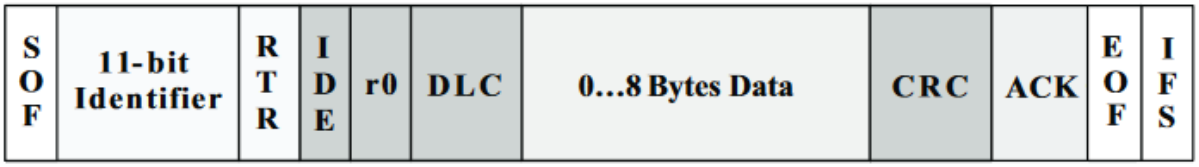


Figure 2-5 Typical CAN message structure [39]

Because of CAN's robustness and the ability of every connected node to individually continue its execution while all are simultaneously receiving system data, it is a proper communication protocol to implement on electronic satellite systems.[41]

2.5.2.3 Serial Peripheral Interface

A Serial Peripheral Interface (SPI) is a synchronous single-master-multi-slave network often used to send data between MCUs and multiple other peripherals, such as SD cards and sensors [42]. It has a similar communication method to that of a UART, except that all devices operate on the same synced clock frequency. The master device can select a slave device to which it can transmit and receive data. This is visualised in Figure 2-6.

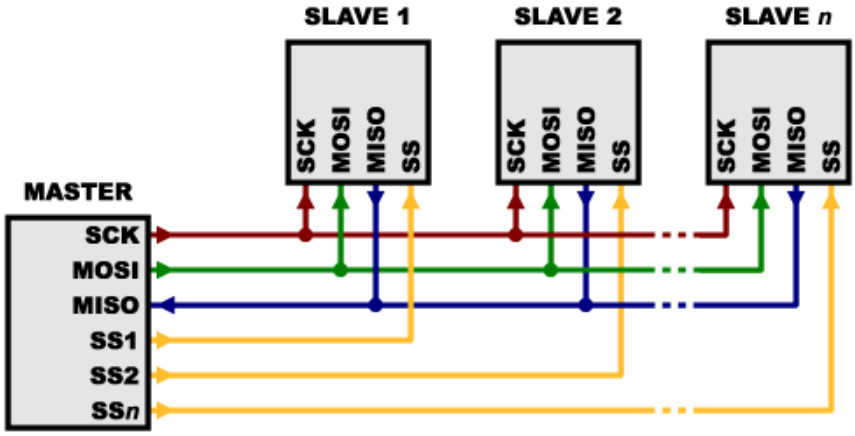


Figure 2-6 SPI connection layout [43]

2.5.2.4 Inter-Integrated Circuit (I2C)

I2C is another communication protocol that allows connections to multiple slave devices from a single master device and numerous master devices to one or more slave devices [44]. This is handy when multiple devices need access to a single peripheral, such as various MCUs, all logging data to the same SD card. Messages consist of different parts, called frames. The address frame allows slave devices to know to which device the message is directed. Figure 2-7 shows a message breakdown.

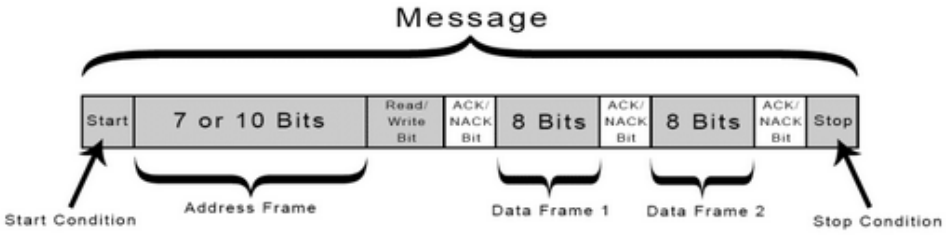


Figure 2-7 Typical I2C message structure [45]

2.5.2.5 Analog-to-Digital Converter

An Analog-to-Digital Converter (ADC) is a peripheral device that converts analogue signals into digital signals for MCUs to process. The digital signal would then represent a discrete version of a measured voltage or current, like battery voltages [46].

As shown in Figure 2-8, the peripheral converts continuous-time, continuous-amplitude signals to discretised signals. This process introduces a small error, whose magnitude is dependent on the ADC resolution.

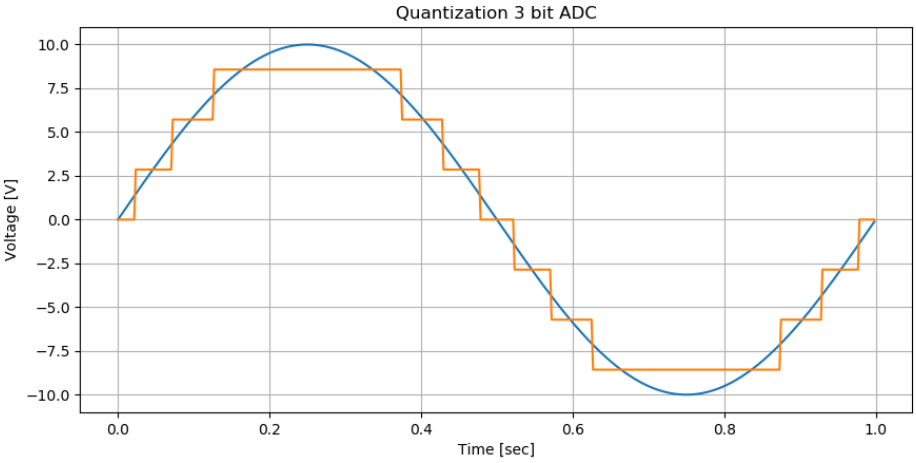


Figure 2-8 Representation of ADC discretization [47]

2.5.3 Expandability to other MCUs

The selected processor for this project, the STM32L452RE, is an Arm Cortex-M4 processor with the Armv7E-M architecture. Arm does not manufacture its own processors but instead licenses the processor architecture to third parties who implement, modify, manufacture and sell the processors.

As a result, the ARM architecture is the foundation for many COTS MCUs, including the embedded MCU in the CubeComputer. The Arm Cortex-M series was specifically optimised to be low-cost and energy-efficient. According to [48], the Arm Cortex-M processors have been embedded in billions of consumer devices, either as dedicated MCUs or "hidden" inside System-on-a-Chip (SoC) chips as I/O, system, sensor, touch screen, smart battery, and power management controllers. Further, the Arm Cortex-M4 is the most widely deployed Cortex-M processor [48].

The Arm infrastructure also includes boundary-scan cells, described in the next section. However, sellers may deliberately disable physical access to prevent consumers or competitive manufactures from gaining access to proprietary firmware or reflashing embedded chips [49].

As long as access can be gained to the boundary-scan cells and a memory map is provided to map those cells to physical peripherals or memory, JTAG, discussed in the next section, can be used to extract or inject data into the MCU. These MCUs should all be compatible with this project by making small adjustments to the interface driver controlling the JTAG programmer to reflect the new memory map.

2.5.4 JTAG

In 1985, when PCB designs became too complex and too small to verify the working of connections between components on the PCB via probes, both to test manufacturing quality and to ensure all components are properly soldered and connected, the Joint Test Access Group implemented what would, within five years, become an industry standard for verifying designs and testing PCBs [50]. This technology was named JTAG, after the group.

Targeted initially towards board-level testing, JTAG specified the use of a dedicated debug port implementing low-overhead access through a serial communications interface that did not need direct external access to a system's address and data busses. This interface connects to a Test Access Port (TAP) on the chip that can access test registers, called boundary-scan cells, presenting chip logic levels and device capabilities of various parts [51], allowing manufacturers to verify the integrity of the manufactured ICs.

These days JTAG is an essential mechanism for embedded system debugging. The system might not have any other debug communication channels, as it is used as the primary means of accessing sub-blocks of integrated circuits. Silicon architectures such as PowerPC, Arm, MIPS, and x86 have embraced JTAG, building entire software debug and instruction and data tracing infrastructures on top of the basic JTAG protocol [51].

Critical to this document is what is known as JTAG boundary scan testing, which provides access to many logic signals of complex ICs, including device pins. These signals are represented in a boundary scan register, accessible through the TAP. In Arm's case, this maps all of the boundary-scan cells in this register to the processor's memory map [52]. Boundary-scan cells can operate in a functional mode where they do not affect the device or a test mode where they disconnect the pins and the device's functional core. This allows users to control the values being driven from enabled devices onto a net and monitor that net's values [50].

In the Arm MCUs, such as the test device for this project, these boundary scan registers are equivalent to the device memory map. By examining this map, it is possible to determine the state of almost all digital device pins in the entire MCU. This includes the states from

individual bit-level latches in memory cells to the logic levels in built-in peripherals to specific peripherals that can be connected externally to the MCU [52].

A JTAG programmer is needed to use these features. Luckily, the development boards sold by STM already come with such a debugger in place, which implements their adaptation of JTAG called ST-Link. This allows users to connect to the debugger via USB. The debugger connects via the TAP to debug and program the board. The ST-Link can also be removed and replaced by a generic JTAG programmer should the need be there. Figure 2-9 shows the ST-Link programmer on the selected MCU.

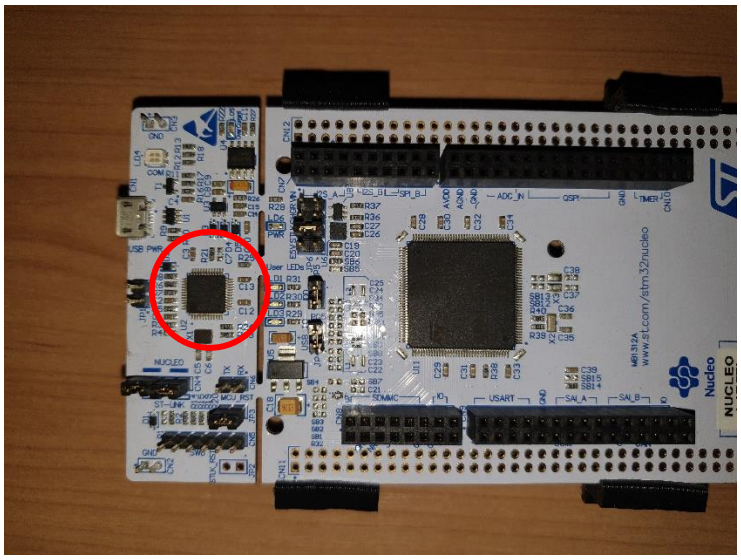


Figure 2-9 Included ST-Link JTAG programmer on MCU development board

2.6 Fault Injection

Fault injection is a technique that can be used as a software testing method where errors are deliberately introduced into the system to characterize fault propagations, test error handling code paths that might otherwise rarely be followed or to ensure the system can withstand or recover from error conditions [81, 82].

SEE testing is a good example of fault injection testing. By using the particle accelerator, testers create an environment where faults are injected at an accelerated rate into the component being tested to determine the component response should these errors occur naturally in-orbit. However, SEE testing is not the only way to inject errors, though it is preferable as it uses the same mechanics to trigger the faults as in-orbit.

If an easier way to induce a similar level of faults can be used in conjunction with SEE testing, it would be possible for testers to prepare for SEE tests by optimising their DUT implementations before SEE testing. Sensitive areas in the DUT can be discovered and mitigated, and mitigation techniques can also be verified. SEE testing can then only be used as a final verification that there are no radiation-specific induced faults, as found in-orbit, that somehow gets missed by the implemented mitigation techniques.

This section investigates some different fault injection techniques and compares their useability for South African testing in conjunction with SEE testing to optimise the time spent in-beam.

2.6.1 Fault Injection Techniques

Faults can be injected into MCUs in various ways. For this project's purposes, only injection methods that function separate from the MCU core will be considered, as the goal is to inject into the core itself. From the MCU's perspective, it should be unaware that any data has suddenly been corrupted.

2.6.1.1 Electromagnetic Fault Injection

Electromagnetic fault injection uses electromagnetic interference to induce faults in electronics. This technique was used in [73] by precisely positioning a 1mm magnetic antenna close to the MCU section to be injected. This antenna then gets energised with pulses with an amplitude range from -200V to 200V and a width that extends from 10ns to 200ns. Their target was an Arm-based Cortex-M3 processor, and testing followed these basic steps as described in their paper: "

- Reset the microcontroller
- Execute the target code
- Send a pulse to the injection antenna
- Interrupt the program execution
- Harvest the microcontroller's internal data "

Data was extracted using Serial Wire Debug (SWD), a debug port for severely pin-limited packages that replaces the normal 5-pin JTAG port with a clock and single bi-directional data pin [74]. This data can then be evaluated to determine if errors were induced and what their effects are.

2.6.1.2 Laser Fault Injection

Laser Fault Injection (LFI) has been proven to be a powerful technique used in fault injection attacks to disrupt internal processes in ICs [75]. LFI uses pulsed lasers to inject faults into running secure devices, usually to retrieve secret information [76].

Experimentation done in [77] used a solid-state Nd:YAG laser source emitting a 1064nm wavelength with a fixed pulse width of 800ps to inject faults into de-lidded Arm MCUs. The MCUs were de-lidded by mechanical means. The SRAM and CPU registers are initialized with an alternating bit pattern. The laser is then used to send precise pulses into the die, following a grid pattern. After injection, the SRAM and CPU registers' data is read out and compared to the known grid pattern to determine error effects. The tested MCUs were

- STM32F051R8T6, an STM32 Arm Cortex-M0 based processor
- STM32F401RBT6, an STM32 Arm Cortex-M4 based processor
- NXP LPC11E14, an NXP Arm Cortex-M0 based processor
- XMC1401-F064F0128, an Infineon Arm Cortex-M0 based processor

2.6.1.3 JTAG Fault Injection

From Chapter 2.5.4, JTAG can physically remove MCU core control from registers or pins, change the state of these pins or registers, and restore control to the MCU. It can also completely halt or resume MCU core operations. Thus, by stopping an MCU core, changing the state of relevant bits, and then continuing the core faults can be injected. Due to the halting of the core, the MCU is unaware of any state changes.

JTAG-based fault injection of core devices is a non-destructive fault injection method that is easy to control [78] and, from the MCU's perspective, can happen in real-time. The approach implemented in [79] used the JTAG interface to inject faults into the instruction register of a MCU and then monitor the error's propagation. This process was executed in the following steps:

- Initialise the system.
- Inject the "sample" (incorrect) instruction into the instruction register.
- Sample the I/O data to a capture region of boundary-scan cells to extract.
- Inject a valid instruction in the instruction register.
- Release the device to continue in normal operation from the valid instruction point.

2.6.2 Sensible Fault Injection for the South African Context

Fault injection can be a practical tool to help local designers perform necessary upset sensitivity and mitigation techniques before testing at iTL. This would be extremely beneficial as scheduling tests at iTL is time-consuming, and testing slots are scarce. However, this is only valid if the fault injection method is more locally available and affordable than SEE testing. Further, if the injection method can be easy to set up, it is more likely to be adopted by the industry, and it can be bundled together with the rest of this project in one package.

Electromagnetic Fault Injection and LFI can inject faults while the DUT system is running. Still, both require complex setups that involve extremely accurate positioning systems to direct the injections into the correct region of the MCU [73, 77]. Further, as the area where faults are being induced is determined through the physical positioning of the injection source relative to the MCU, in-depth knowledge is needed of the physical layout of the MCU. For most COTS MCUs, this information is not available to the public. Also, both injection methods inject into MCU areas rather than just specific cells. This makes it hard to pinpoint components like core registers due to their small size relative to the injection source, even if their exact position is known. Both Electromagnetic Fault Injection and LFI require big setups, with LFI being one of the most expensive techniques to set up and use to inject faults with [80].

JTAG injection can only inject faults in pseudo-real-time by controlling the execution flow of the MCU. However, it is incredibly affordable and already integrated into almost all Arm processors. As seen in Chapter 2.5.4, the selected MCU for this project is available on a development board that already comes with a JTAG debugger attached and can allow for simultaneous communication between a PC and the MCU through both JTAG and UART. JTAG injection also gives precise control over which individual registers, and even specific bits, can be corrupted and can be used to monitor the effects of the injected errors. This can not be done through either Electromagnetic Fault Injection of LFI, and a secondary process, often JTAG as seen in the next section, is used to extract the data.

It would be the most sensible for testing in South Africa to combine SEE testing with JTAG fault injection and data extraction.

2.7 MCU SEE Testing

The problem of flipped bits and corrupted data is not limited to the space environment. As processor designs become more complex, they become more vulnerable to environmental disturbances, SEE, latent defects from the manufacturing process, and verification

inefficiencies that allow design bugs to propagate into the system. Similarly, SEE does not only occur in space. Cisco has had several cases where SEUs are claimed to cause transmission errors in their network routers and even has a guide on how to implement a workaround for their 12000 line cards [53].

This section will investigate relevant previous MCU SEE testing methods to identify the required test suite interfaces.

2.7.1 Previous SEE MCU testing

COTS MCUs are gaining popularity for use in satellite designs. They might not be used as the central onboard controller but are used to control secondary processes. As such, several researchers have started investigating SEE effects in COTS MCUs.

Proton irradiation was done in [54]-[56] for SEU characterization of the Pentium (R) MMX, Pentium (R) II and Celeron microprocessors between 2000 and 2009. In 2001, [57] investigated SEU in the PowerPC750 MCU. [57] also made use of JTAG to be able to perform testing on the DUT cache. In 2009 the clock and reset transients of a 90nm RHBD single-core Tilera processor was tested [58].

In 2014, [59] investigated SEU in low-cost, low-power COTS MCUs, including the 16-bit MCU Texas Instruments (TI) MSP430, the dual redundant Arm Cortex-R4 TI Hercules, the Arm Cortex-M4F TI Stellaris, the Arm Cortex-M4 TI Tiva and the Arm Cortex-A9 MPCore Xilinx Zynq. The authors also indicated a particular interest in Arm MCUs as they provide low-power, low-cost alternatives for general computing.

For SEE testing in [60], a compendium of TID, neutron, proton, and heavy-ion SEE tests on various satellite electronics, including the Arm-based NXP LPC2148 and STM32 F417IGH6 processors, were done. The STM32 MCU is similar to the MCU chosen as the test subject for this project. The processor showed SEUs and two SELs during neutron testing, SEUs during proton testing and non-destructive SEL with all heavy ions tested with a LET > 2.19 MeV-cm^2/mg.

For the SEE testing in [61], JTAG was used to verify the reliability of software algorithms and software-based mitigation techniques in DSPs, where JTAG could be used to access the memory of the entire DUT. The authors claim that, for their tests, JTAG was the easiest way to access register and cache space, opposed to previously used assembly codes in [62], a guideline for ground radiation testing for space radiation. Their results did show, however, that USB connection speed of the JTAG programmer influenced their results.

The research done in [59] – [61] was all part of research at Los Alamos National Laboratory, connected with other research groups. They show a clear focus towards testing COTS MCUs, with a particular interest in Arm-based processors. The work done in [57] and [61] also indicates that JTAG manipulation of the boundary scan register can be used as a viable method to interface with DUTs during testing.

2.7.2 MCU Testing Approaches

In [56], the processor would execute a program and compare the registers with known values every computational iteration. If the values do not match, it reports that an upset was found. Several specific tests were performed:

- An Arithmetic Logic Unit Test performed numerical factorization involving integer multiplication, division, and addition. Every step of the factorization includes comparing the general-purpose register values to known correct values, and SEUs are reported if the compared values do not correlate.
- A Floating-Point Unit Test calculated an approximation of the value of π through addition, multiplication, division, and subtraction. Again, at each step, the floating-point register stack results are compared to known correct values.
- A Register Test loads some of the registers with 0x55 and loops, checking that the registers contain the correct values.
- A Cache Test that is similar to the Register Test, except applied to the L1 cache.

In [57], the DUT also mostly performed self-analysis and used the JTAG interface to investigate the contents of the L1 data and instruction caches that were otherwise inaccessible. The following tests were performed:

- The "do little" Test had the DUT perform a single-instruction infinite loop that gets briefly interrupted every half-second to save a register snapshot to the physical memory. After irradiation, the snapshots are downloaded and analysed for irregularities.
- The "pin wiggler" Test had the DUT perform a self-inspection of one of its internal memory arrays or register files. If an error is found, an address pin is toggled. External counters monitored this pin to give live feedback on radiation irregularities.
- The Cache Test initialized the L1 data and instruction caches in specified conditions and then disabled them before the irradiation started. A clearly recognized pattern, distinctly different from that of the cache, was placed in the memory space covered by the cache. These areas were then irradiated and afterwards extracted through JTAG and compared with the original patterns.

- Additional test programs for the FPU to determine transient logic errors. No further specifics were given about their implementation.

The Los Alamos National Laboratory also has several MCU benchmarks [63] that form part of their MCU tests. The DUTs run the benchmarks and give YAML parsable text output via UART to be monitored externally. This output contains information about the errors detected during the execution of the benchmark. At the time of this writing, four algorithms are publicly available:

- Advanced Encryption Standard: This code executes a 128-AES code from Texas Instruments using the National Institute of Standards and Technology standards.
- Cache Test: A program for instrumenting SRAM memory blocks or caches by testing the Foura mostly zeros memory pattern and implements a sum of memory array elements to check for transients in logic.
- Matrix Multiply: A program for calculating matrix multiplies and takes approximately the same memory as the cache test.
- Quicksort: A program for testing quicksort. Data is randomly generated and placed into an array, and the inputs change every few seconds in a repeatable pattern.

In [59], these benchmarks were used in tests in combination with watchdogs and hard resets, along with other benchmarks not yet released to the public. The authors of [64] also describe such an approach and mentions exploiting special purpose hardware modules called watchdog processors to gain access to and monitor the memory. However, they also note watchdogs do not detect latent faults in the MCU or faults in the register bank. This shows that it is possible to use built-in hardware to expand monitoring for SEE, should it be available.

It is also possible to use hardware-based detection and mitigation techniques, but they usually change the original architecture by adding logic redundancy, error-correcting codes, and majority voters[64]. This does somewhat defeat the purpose of using COTS components in the first place.

2.8 Summary

This chapter introduced various concepts, mechanics, technologies and research methods relating to SEE testing of COTS MCUs. The effects of ionising radiation were investigated, along with a testing approach for SEE tests specific to iTL in South Africa. The STM32L452RE MCU was selected as the test subject for this project, and various MCU technologies and testing approached were introduced. In the next chapter these concepts will be combined and reworked into a concept design for a SEE MCU test suite.

Chapter 3
Concept Design

As described in Chapter 2, ionising radiation testing can be extremely beneficial in prolonging the life expectancy of MCUs used in satellite systems. For this project, identifying the specific type of SEE will not be the main focus. Nor is the focus on getting an SEU cross-section for one particular processor or testing a specific mitigation techniques' efficiency. The focus will be on designing a test suit that allows testers to better prepare for SEE tests, verify their particular applied mitigation techniques before testing, help them determine cross-sections and SEU sensitivity for a range of MCUs, and boost their efficiency while performing SEE tests.

This chapter presents the theoretical implementation of a MCU SEE test suite, focussing on the design drivers and needs, various methods to communicate with the DUT, typical peripheral tests that can be used to verify the test suite functionality, and finally the iterative approach that was used to implement these designs.

3.1 SEE Test Suite Design Drivers

To achieve the goals mentioned above, a test suite is developed and the following requirements and limitations drive the design thereof:

- The test suite must be used for easy integration into the SEE testing environment, focusing on iTL.
- The test suite must be compatible with as wide a range of MCUs as practical, to not restrict the designer's options.
- The test suite must help designers prepare for SEE testing, to aid in the effective use of beamtime when testing.

- The test suite must be intuitive and straightforward to use, to limit the chance of mistake on the test day.
- The test suite must be versatile and easily adjustable to accommodate different onboard test approaches.
- The test suite must limit its internal overhead usage in the DUT to not interfere with different testing approaches.

A common problem with SEE testing lies in the extraction of errors. Another common problem is the lack of testing facilities and beamtime to perform SEE tests. When preparing for a test, the tester must ensure that the test will not be in vain as it could be months before another test can be scheduled. When performing SEE tests, it is important that the tester can test efficiently and precisely.

Before testing, the tester should be able to simulate SEUs in the DUT to analyse the DUT's reaction to upsets in various locations. If simulated errors injected into particular parts do not affect the system's overall performance, the tester knows that mitigation is less of a priority in those areas. In contrast, if the injected error heavily impacts the DUT, the tester knows that area is vulnerable to corrupted data and can choose to apply a mitigation strategy to strengthen its SEU resistivity. This does not give information on which areas SEUs are more likely to occur in, but it shows which areas will be the most affected if an SEU occurs there.

During testing, the tester should be able to interface the DUT and the test it will be running through the test suite to track the DUT status. The test suite should be able to identify SEUs in as much of the processor as possible to aid the onboard firmware. It should also be able to upload new firmware and power-cycle the DUT if needed.

The specific test executed on the DUT during SEE testing is the responsibility of the tester to implement. The test suite will then serve as an interface between the DUT and the tester and strive to simplify the test experience.

Alternatively, an approach such as the “do little” test from Chapter 2.6.2 can be implemented where data is only extracted once irradiation is completed. Unfortunately, this would leave the tester inherently blind to see critical errors or SEFIs occur in real-time if the DUT is executing code. Usually, it is impractical to extract all the available information from the DUT in real-time as it can be time-consuming compared to the rest of the DUT's execution. It would be much more viable to use the MCU to determine which areas stop functioning, and only extract the DUT data when a significant number of errors are reported, or a loss of functionality is detected.

Defective peripherals in the DUT will need to be detected using different methods depending on the peripheral's operation and will be discussed in the following sections. Once the DUT detects a loss of functionality, the test station can halt the program, extract all relevant data, and resume execution or reset the DUT.

For the POC in Chapter 4, no radiation tests will be done, as the goal is only to establish whether there is a viable way to extract or inject data when needed.

3.2 Interfacing the DUT

3.2.1 DUT communication options

Communication between the tester and the DUT can happen in various ways. This section investigates different raw data reading and writing methods from the available communication channels between the DUT and the test station, namely UART and JTAG. Preference will be given to the method with the least overhead, affecting the executing firmware as little as possible.

3.2.1.1 UART

One of the most common ways for programmers and engineers to transfer data to and from an MCU is by using one of the MCU UARTs for serial communication. It is quick and easy to implement and integrates effectively into a debugging workflow. It can be convenient for most normal debug operations, but it is also slow at transferring large data chunks.

Serial communication could be used to read the MCU data by broadcasting all the data in the areas of interest or transmitting a summary of errors after a self-diagnosis by the MCU. However, transferring data over serial is slow and greatly limits real-time effectiveness when sending large messages. Depending on the MCU size, it could take very long to transfer the state of every bit and pin accessible. It also creates overhead and slows down other operations the MCU might need to execute.

Further, as this implementation would make it part of the MCU workflow and require the MCU to scan the investigated areas itself, it is untrustworthy. It will be directly exposed to the same conditions as the rest of the MCU, and the UART communication will disrupt and dominate the execution cycles.

Even if it is easy to implement, an optimal approach when using the UART to verify flight-ready software elements is only to report the minimum data needed, and significant extractions are only done when required to reduce overhead. Using a UART to read large

quantities of data will always affect the execution of the host system due to the operation's overhead, which can muddle the SEE sensitivity results of the rest of the system.

An argument can be made that the additional overhead can be tolerated. If the test's goal is to estimate how long the DUT will survive LEO while executing arbitrary firmware, then the overhead of using the UART can be accounted for in the test procedure. Further, a test that uses firmware to do heavy self-diagnosis to the extent that every possible detected error has to be reported to an external device is most likely not running real-world flight-ready firmware anyway.

This is the case for the benchmarks from [63]. They were specifically designed to give effective UART output, and the entire benchmark design had a strong focus to try and minimize the effect of the UART transmission overhead compared to the actual benchmark execution. These benchmarks give no direct indication of a specific piece of flight firmware's sensitivity to SEU, but they provide a good estimation of specific peripheral response in general. Thus, using UART communication to extract data from the DUT can be done if care is taken to minimize the effect of the transmission overhead.

Injecting data through UART by telling the MCU in what location to corrupt data is not practical. There is little control of when in the execution cycle the data will change, and not all addresses are available to be changed by the DUT. However, transmitting data through the UART to interact with the program executing on the DUT is common practice and could easily be used to enter specific states in the DUT program or control the program flow.

3.2.1.2 JTAG

JTAG, as technology, is discussed in Chapter 2.5.4. It gives a dedicated debug port that can be used to access the boundary register cells to view the state of all device registers, memory latches, and pin states. As such, it can be used to inspect the condition of the entire memory map. It can also be used to transfer data into non-volatile device memory for device programming or into volatile memory if the memory region is initialised. Several debug functions, such as halting, stepping, and resetting the MCU core, are also available. As stated in Chapter 2.6.2, this will be the primary method of injecting data into the MCU.

To use standardised JTAG on the chosen MCU requires that the ST-Link be disconnected from the development board JTAG interface. A typical JTAG layout is displayed in Figure 3-1. The pins are:

- TDI – Test Data in
- TDO – Test Data Out

- TCK – Test Clock
- TMS – Test Mode Select

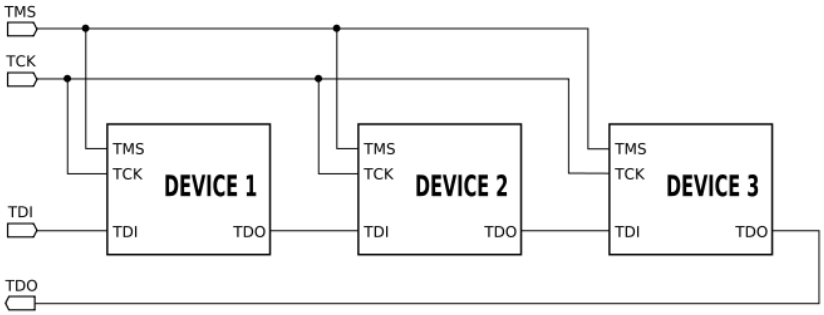


Figure 3-1 Standard JTAG connection and daisy chaining.

3.2.1.3 ST-Link

ST-Link is STM's modified version of a JTAG programmer and ships with the development board. It was created to use the USB full-speed interface to communicate with several IDEs, such as Atollic, Keil, and TrueStudio [65]. It also contains a UART-to-serial converter that allows the programming device to communicate with an onboard UART on the MCU for basic communication and debugging purposes.

It is slightly slower than JTAG but has the advantage of being plug-and-play on both the development boards and the IDEs. Without knowledge of the DUT or the boundary scan registers' internal mechanics, it can be nearly impossible to use JTAG to communicate with MCUs. Luckily STM provides both the memory map and drivers that allow seamless communication via JTAG for debugging purposes, if using ST-Link.

STM also provides a command-line terminal interface utility, called ST-Link CLI, that can be used to automate the programming and debugging of MCUs. This can be controlled programmatically by sending the correct commands to the terminal via Python scripting, allowing for the automatic scans of the memory map.

The ST-Link CLI can also be used to change, or inject, upsets into the MCU. Since the core can be halted, the upsets injected and the core resumed, the MCU should experience a similar situation to SEU. Specifically, that data has suddenly changed when it should not have from external factors.

3.2.1.4 Sensible application of communication methods

As seen in Chapter 2.7.2, both JTAG and UART has been used to extract data from devices to verify if SEUs occurred. In certain cases, especially when retrieving cache data, only JTAG could be used. Thus, it makes sense to design the test suite to accommodate both and leave the specific implementation to the tester. However, for fault injection, the UART is not a viable option. Luckily JTAG has the capability to inject errors into the MCU discretely.

As the chosen MCU comes with the ST-Link connected on the development board, and both UART and JTAG can be used by using the ST-Link programmer, it will also be the programmer used during the testing of this project. However, should there be a situation where the ST-Link needs to be switched with a more generic JTAG programmer, only a driver change will be required.

3.2.2 Onboard testing

3.2.2.1 Peripheral testing

As mentioned in the previous chapter, several peripherals can easily be tested for functionality by the DUT itself. Many of these tests could also be done for TID testing to verify component functionality, as described in Chapter 2.2. This section will look at ways to implement these tests, should the need arise for a tester to focus on the effect of SEE on specific peripherals. If these tests show some, or complete, loss of functionality of the peripheral, such as unsynced clock signals, message or latch corruption, floating or stuck A/D readings, or even complete peripheral failure, then more advanced testing methods can be investigated regarding the specific peripheral.

3.2.2.1.1 Flash memory

The flash memory data integrity can be tested by the chosen MCU's built-in Cyclic Redundancy Check (CRC) Calculation Unit. Should irregularities occur, the entire memory contents can be read out via the data extraction discussed in Chapter 3.2.1.3 and compared to a known valid state of the memory to find the error. This could be especially handy to also detect errors in areas of the memory seldom used and where changes could have little effect and take time to propagate.

CRC can be used for error correction, but in order to determine SEU locations, it is recommended not to use the error correction. Once a model is built from SEU location probabilities, the error correction can also be switched on to determine its effectiveness.

3.2.2.1.2 Analogue-to-Digital Converter

To test the Analogue-to-Digital Converter (ADC) a simple predetermined voltage-divider voltage can be continuously sampled and compared with the known value. The voltage divider can also be powered by the MCU I/O pins, allowing the MCU to toggle between the known voltage and ground.

This setup is displayed in Figure 3-2. Switching it on or off helps to detect if the ADC is stuck at a fixed value. V_{in} is driven by one of the IO pins. If the resistance at Z_1 equals the resistance at Z_2 V_{out} would be half of V_{in}. Thus, if V_{in} is powered, a constant value is expected at V_{out}, and if V_{in} is not powered, V_{out} should be ground.

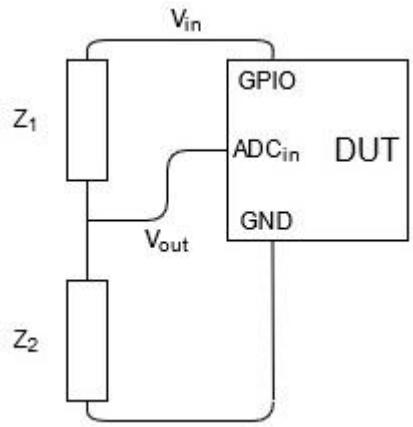


Figure 3-2 ADC voltage divider

3.2.2.1.3 Communication peripherals

There are various ways to test a communication peripheral's functionality, and a low-level test could be set up for each. However, a low-level test is not needed, as the only interest is to determine when a peripheral lost functionality. If a loss in functionality is detected, the peripheral registers can be read to determine where the fault occurred via JTAG.

Specific communication peripherals can be tested on their own. For example, a UART receiver could be connected to the transmitter of that same UART. When a message is transmitted, the same message should also be received. For other peripherals, like CAN, there must be more than one device connected to the communication bus. Otherwise, the peripheral cannot function. The easiest way, especially when using MCUs like the STM32-range with a sizeable peripheral set, is to have multiples of the same peripheral set up and connected. Not only does this double the exposure the type of peripheral is experiencing to

injected upsets, but most peripherals can monitor the communication bus to verify its integrity. If a UART receiver is pulled low for longer than the span of a character, it triggers a break condition. For CAN bus, the standard even specifies a break condition if there appears to be a physical line break if only two CAN devices are on a network and one stops functioning.

There are various types of break conditions, and implementations differ between peripherals and MCUs. Still, in most cases, the peripheral will have error registers or flags inside an internal register that can trigger an exception when a break condition happens to allow the MCU to see that at least one of the connected peripherals is experiencing a problem. These registers or flags could then be extracted via JTAG.

When running particle radiation tests and not just injecting errors, all peripherals are exposed and could be vulnerable to SEU. The chances of a crippling SEU soft error occurring on two connected peripherals on the same device should be relatively low. Still, the connected peripherals may share resources as they are on the same device. As such, should a shared resource, like a clock line, be affected by an SEU or SET, it could potentially disable both peripherals. For this reason, it might be better to have a similar, shielded MCU to act as the mirror peripherals. This way, the peripheral integrity can be verified independently from the DUT. Having a separate device to test communications also allows the user to compare if and how peripherals degrade and gives them the freedom to use and try any combination of peripheral devices.

It can somewhat simplify the process if assuming that the exact error does not have to be detected when an SEU occurs, but only that one occurred and its functional severity. Should a message not be correctly transmitted or received, or corrupted during transmission, the system should only have to report that something, somewhere, went wrong. The test station can then extract the register values to determine where the SEU occurred.

For the POC, since it will not test in a hostile environment, the peripherals will be connected on the DUT to verify the workings of the SPI, I2C, CAN, and UART devices, except for the UART communication to the test station, which the test station itself can monitor. For the chosen MCU, these could all be initialised without pin or clock clashes.

3.2.2.2 Optimizing data extraction for speed

As mentioned in Chapter 3.2.1.1, outputting large amounts of data through the UART to the test station can be time-intensive and impact the DUT's ability to detect SEU as they occur. Using JTAG to extract large chunks of memory states might also not be very fast.

However, there is still a need for near real-time feedback from the DUT to allow the tester to react to functionality losses.

Since the tester would want to maximise their time with the beam, one approach could be to discard the more in-depth accuracy of where SEU occur and instead focus on the DUTs durability. This way, the focus is not on determining SEU sensitivity, but rather on SEFI sensitivity. Practically, the SEFI rate will be a significant factor in deciding to use the DUT for a LEO mission, especially if the DUT's role will not be critical.

Mitigation can be applied to the board to restore as many SEUs as possible, and data can periodically be extracted by JTAG extraction without stalling the core. Rather than remove large areas, only small sectors can be read, and the SEU can be found externally by processing the extracted data.

This allows for the processor to be exercised fully without much of the SEE test overhead and to spend the maximum time in the beam. This does limit the visibility of SEU but can still indicate a DUTs in-orbit lifespan. Another downside to this approach is that the collected data will be much less. To make a probabilistic guess to the DUT's resistance to SEFI, several runs will be needed to get the minimum number of errors to be statistically significant, which might increase testing time anyway.

3.2.2.3 Optimizing data extraction for accuracy

One of the reasons to perform SEE testing is to find out which areas of the DUT is sensitive to SEE. Mitigation can then be applied more heavily on those sectors, while areas less sensitive to SEE can have less mitigation applied.

To identify sensitive areas, the DUT can be exposed for a short interval, then all data can be fully extracted via JTAG, and the beam is reactivated. This process repeats until the SEE testing is done. This would allow for large data extractions that can precisely predict the SEU cross-section of various DUT elements for the specified beam energy. This does mean that short bursts of irradiation might be overshadowed by long extraction times if the data extraction process is not quick enough.

3.2.2.4 Balancing speed and accuracy

A balanced approach between speed and accuracy will, in most scenarios, be the best approach to get the most information while still spending enough time in the beam. The easiest way to do this is to have the DUT monitor high-level functionality and output its

status through the UART. Once it has detected enough of a loss of functionality a deep extraction through JTAG can pinpoint where the remaining upsets occurred.

Some upsets will be lost or masked, and some might have propagated to create various other errors, but this way testing extends the DUTs time in the beam, and injection can be used afterwards to replicate the SEE patterns to test if it induces similar results. This should be good enough for most mitigation schemes, but also gives an estimate of the durability and life expectancy of the DUT in LEO.

3.2.2.5 Interfacing with the onboard tests

As seen in the tests in Chapter 2.7.2, two main interfacing methods are typically used with the DUT while being irradiated. The device UART gives real-time feedback about the current condition of the DUT, as with the benchmarks in [63], or the DUT is irradiated, and afterwards, sections of memory are extracted through JTAG which it can be compared to a known correct state and errors can be calculated by analysing the differences between the known state and the extracted one, such as with the cache test in [57]. Some tests also used a combination of these two options.

To build a test suite that can be used to enhance SEE testing of MCUs both of these communication methods, at minimum, need to be compatible with the system. The tester will need to be able to dictate when and where to extract data through JTAG. If the DUT test uses UART output, the system should parse this output to react if certain messages are detected automatically.

For example, the benchmarks in [63] give YAML parsable output over the UART that list where errors are found. Should a significant number of errors be detected the test could be stopped, the beam disabled, the core halted, and a full data extraction could be performed via JTAG. This allows real-time tracking of the error score and gives in-depth information regarding exactly which bits are incorrect and can be used to determine biases, such as if bits flip easier from high to low or vice versa.

Given that different tests might have other data sent over the UART, the tester should be able to modify the way the system parses this data and reacts to it.

3.3 Test Suite Adaptability

To accommodate multiple MCUs and a variety of test scenarios the test suite must be designed in such a way that a tester can easily modify certain sections to suit their needs

without having to alter the entire program or setup. This section will briefly mention some relative design points relating to this customizability.

3.3.1 Programming Languages

To maximise the ability of testers to be able to customize the test suite to suit their needs the test suite must be programmed from languages that are popular in the data science community, easy to read to aid in code modification, easy to install and get an environment set up, and functional enough to achieve the desired goals. To this end, Python was selected as the primary programming language.

Node.js is a server-side platform that uses JavaScript as programming language and is also used later in the development process to enable enhanced, fluent and easier-to-operate GUIs. GUI screens are made with basic implementations of standard HTML, CSS and JavaScript and are very easy to customize or change. This is, however, optional. A Python-based GUI can be sufficient and easily swapped in by the tester.

Both JavaScript and Python are extremely popular in South Africa, and both have ample documentation online for most imaginable scenarios. However, it is important to note that the JavaScript-based GUI is just used for ease-of-use, not for any sort of calculations as it does not handle floating-point numbers well. All core functionality is implemented in Python.

3.3.2 Test Configurability

It is important for the test suite to adapt to specific test needs. As such, the development of all critical parts of the system will be independent. This will allow testers to change certain modules without affecting the rest of the system. All modules communicate with each other with basic APIs. The final version of the APIs at the completion of the project is given in Appendix B.

Should the benchmarks from [63] be used, the UART module could then be configured to automatically filter the DUTs YAML output, and automatically start a full data extraction via JTAG once a certain number of errors have been reported. Should there be a need for the test station to operate SEL detection hardware, the *Devices* module could be updated to drive the hardware. Should the user want a different communication protocol, the *StateMachine* module can be modified. Should there be a need to use a custom driver for a specific JTAG programmer or MCU the *Driver* module can be replaced.

This modular approach uses modules that can easily be modified or entirely replaced to adapt to any testing need, as long as all modules still adhere to the relevant APIs.

3.3.3 JTAG Drivers

The JTAG driver, in this application, is the piece of code that is used by the Python script to drive the JTAG programmer. Different JTAG programmers will need different drivers to work. As mentioned in the previous section, as long as all APIs are adhered to, the driver can be a custom-designed one just for a specific DUT. It can be a more enhanced driver capable of driving multiple different types of MCUs. It can even be official software from the DUT manufacturer that is manipulated through calls to the operating system, as is the case for the POC in Chapter 4.

3.3.4 Hardware Adaptations

For some tests, it might be acceptable to simply connect the DUT to the test station and start testing. However, especially for higher energies, it becomes important to have safety electronics to protect the DUT from damage caused by destructive SEE. For the SEE test in Chapter 5, with particle energy of 66MeV/particle, switches were added to the USB cables that lead to the DUTs to be able to power-cycle them should the need arise. For tests with particle energies over 180MeV, SEL becomes a significant concern and SEL detection and prevention circuits are needed. It is also better to keep the DUT unpowered while not being irradiated as that prolongs its testing lifespan.

The test station should have I/O pins that can be connected to whatever hardware adaptations are deemed necessary and are controlled by the test station software. As with the other modules, the method of operation is completely controllable and modifiable by the users.

3.4 Testing Station

As humans cannot be near the vault while the beam is being delivered, remote communication with the DUT is needed. However, both UART and JTAG programmers connect to a host system through USB, which cannot easily communicate over long distances.

For USB 2.0, USB cables are restricted to a maximum length of 5m, as per the USB 2.0 specification. For USB 3.0 no absolute maximum length is specified, requiring only that cables meet electrical specifications. Practically, too long lengths of wire will not reliably transfer data due to power loss and parasitic effects. For AWG 26 wires in copper cabling, the maximum practical length is only 3m [72].

As seen in Chapter 2.4, the vaults at iTL have built-in ethernet cables. This allows communication between devices inside the vault and in the control room. Thus, a testing

station is needed that can convert the communication between ethernet and what the DUT requires. This test station can also be responsible for controlling any safety hardware connected to the DUT.

3.5 Monitoring Station

The monitoring station is from where the tester monitors and controls the DUT. It is managed directly by the user and communicates with the test station over ethernet. It needs a good user interface to allow for ease of operation and must be able to parse incoming data for SEU.

3.6 Iterative Design Approach

With the desired test suite laid out in this chapter, an iterative design approach can now be implemented to build the test suite to execute its desired functionality. This process was broken into three steps:

- Proof of Concept (POC): A POC is built to determine if the JTAG interface can practically be used to extract and inject data into the MCU. This version is not tested in a SEE test.
- Iteration 2: The biggest problems of the MCU are addressed: interface speed and lack of a test station. The GUI is also adjusted to better suit SEE testing environments. This version is then tested in a real-life SEE test at iTL to analyse its performance.
- Iteration 3: UART communication is added, and existing features that did not perform to satisfactory levels, such as the user interface and network reliability, are either improved or completely overhauled. A method of simple automation is also added to aid in both device monitoring and fault injection. This version still needs to be tested in a real-life SEE environment.

Chapter 4
Iteration 1: Proof of Concept

The previous chapter explored the various aspects needed for the test suite. From these aspects a practical test setup can be made to verify the validity of the design approach. Before research time is spent developing a full system, it is prudent to first test if the core methodology will be practically usable, and if so can further be developed into a more usable program.

The Proof of Concept (POC) is designed to test the core functionality of this project. This chapter describes the implementation of this stage of the design.

4.1 Description of Iteration

As seen in Chapter 3.2.1, it is often impractical to extract all the available information from the DUT in real-time. It would be much more viable to use the MCU to determine which areas stop functioning and only extract the DUT data when a significant number of errors are reported, or a loss of functionality is detected. Alternatively, an approach such as the "do little" test from Chapter 2.6.2 can be implemented where data is only extracted once irradiation is completed. Unfortunately, this would leave the tester inherently blind to see critical errors or SEFIs occur in real-time if no other communication with the DUT is established.

For the POC, no radiation tests will be done, as the goal is only to establish whether there is a viable way to extract or inject data when needed. If it is found that the DUT cannot reliably be interfaced through JTAG, a different approach will need to be implemented.

This iteration consists out of a testing station, which will be a standard PC, connected via USB to the ST-Link debugger, which is directly connected to the MCU through the development board via JTAG and UART. The ST-Link debugger manages the conversion between UART and JTAG signals for the MCU to serial for the PC. This is shown in Figure 4-1.

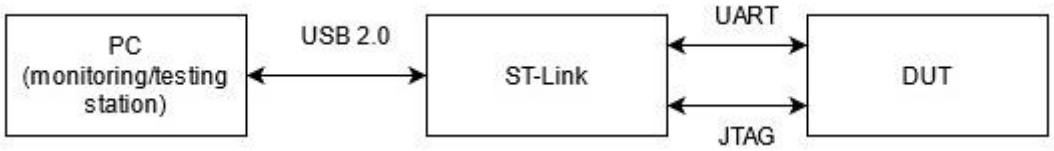


Figure 4-1 The connections between the PC and the DUT

Communication between the PC and the ST-Link debugger will be managed through the use of a Python script that will consist of three threads: a Graphical User Interface (GUI), a virtual COM port listener, and the ST-Link controller. The virtual COM port listener listens for UART transmissions from the DUT, while the ST-Link controller can send commands over the virtual COM port to the DUT UART and invoke the ST-Link CLI though the terminal.

4.2 DUT configuration

Chapter 2.2 lists the more commonly used peripherals for satellite use. To accommodate these peripherals, the DUT can be flashed with firmware initialising the peripherals and performing fundamental analysis to check whether they appear functional.

The DUT will determine defective areas by running a CRC check for flash memory validation. It can run multiple timers and compare the time difference to see if one stalled. By reading a fixed, pre-determined voltage with the ADC, its functionality can be verified. Having one UART, one SPI, one CAN, and one I2C communicating with another identical MCU that can validate the data sent and received could be used to detect errors in their operation.

4.3 Test Station

The DUT is connected to a PC, which acts as the test station, through serial communication over UART and JTAG. A Python script on the PC allows the virtual com port connected to the DUT to be monitored and also operates the communication driver for the JTAG.

In this design stage, the testing station is also the monitoring station, connecting to the JTAG port on the DUT to extract and inject data and the UART on the DUT to monitor the system functionality.

4.3.1 UART communication

Regular communication between the DUT and the PC is vital and needs to be continuously verified to ensure the DUT is still functional. Should a SEFI occur or the UART break down, the monitoring software will not be able to communicate with the DUT.

For serial communication between the DUT and PC a certain pattern was set up to regulate communications:

$$<cmd><len><message>&&

The commands are listed in Table 4-1, with <cmd> being the command type, <len> the length of the message, and <message> the message itself.

Table 4-1 POC UART communication protocol

Command	Process	Example
1	Communication verification: Transmits a number as the message and expects to receive an incremented number.	$$1210&& Cmd=1, Len=2, Message=10 Expects $$1211&& in return
2	ADC readout: Requests the value read by the ADC on the MCU.	$$20&& Cmd=2, Len=0, Message=N/A
3	Peripheral Status: Requests the status of a specified peripheral. Message: 0 -> UART2 Message: 1 -> CAN1 Message: 2 -> SPI1 Message: 3 -> I2C2 Message: 4 -> I2C3 Message: 5 -> SPI3 Message: 6 -> ALL	$$316&& Cmd=3 Len=1, Message=6 MCU sends status of all peripherals to PC
4	Automatic mode: Continuously scan for and report defective sections.	$$40&& Cmd=4, Len=0, Message=N/A

The DUT should always respond to a command from the PC. To verify the DUT is functional, the PC periodically sends a <cmd=1> with an integer to the DUT, which then returns an incremented integer, confirming that the core functionality is still active. Should the DUT not give a valid response, the MCU core can be halted via JTAG, and the memory dumped to the PC to determine where the error occurred. The DUT can then be restarted. If the MCU does not want to initialise correctly, an error may have occurred in the firmware. The monitoring software then validates the firmware on the DUT and reflashes it if necessary.

Commands with <cmd=2> and <cmd=3> allow users to manually initialize tests for the peripherals as described in Chapter 3.2.2. Commands with <cmd=4> allows the DUT to continuously scan all sections for functionality loss.

4.3.2 Implementation of fault detection and injection

To extract and inject data from the MCU, the ST-Link protocol and programmer was used. On the test station PC, the ST-Link CLI interface was used to communicate with the JTAG TAP via the ST-Link programmer on the development board. Most ST32 boards will support ST-Link, but the ST-Link CLI also functions with certain JTAG programmers enabling support for a broad range of processors. The process implemented mimics the action of a user manually opening a terminal window, entering the correct call to ST-Link CLI, and then reading the output.

A Python library was developed to interface between a Graphical User Interface (GUI) controlled by a user and up to two MCUs at once. This allows easy operation by the user. Connecting up to two MCUs allows the operator to use one as a shielded communications monitor if the POC should ever be in a radiation test environment. This library also detects the serial communication from the MCUs UART over virtual com ports and uses the communication protocol as described above. The ST-Link Controller part, which controls the JTAG communication and UART transmits from the PC to the DUT is shown in Figure 4-2.

4.3.2.1 Python library

The developed library uses the command line to call the ST-Link CLI with the correct parameters to achieve either a data read or a data write, or manipulate the MCU core. This is the same as if the user were to manually enter these commands into the command line to run the ST-Link CLI application. It dramatically increases the speed and accuracy as it operates much faster than a human operator could. This is shown in Figure 4-2.

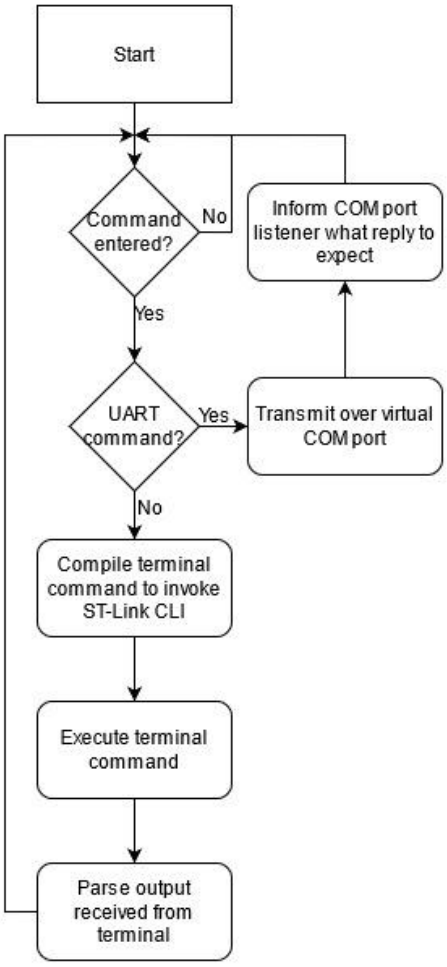


Figure 4-2 The operation of the ST-Link controller module

The core functionality of the library allows users to quickly search for all available devices, connect with those devices, read single bytes or 32-bit words, dump bigger sections of memory into files, write to the MCU memory map in single bytes or 32-bit words, halt, reset and step the MCU core, set and clear breakpoints, read or write to the MCU core registers (device must be halted), and flash new firmware. It also lists all available com ports, connects with selected com ports, transmits messages, and monitors the connections for incoming messages.

Since one of the goals of the POC was to determine if this system is fast enough to be used in a real-life radiation testing environment, each function also has a timed version which can be accessed by calling "_time_<functionName>(<parameters>)", which executes the default function and returns the time the execution took.

4.3.2.2 GUI

Repeatedly typing commands or manually calling functions can be a very impractical way to do SEU testing. As seen in Chapter 2.4, should this system ever be used for a particle radiation test, there is a big need for effective and accurate testing. As such a GUI was developed to allow users to easily operate the developed library mentioned in the previous section.

The GUI makes it easier for the user to monitor the MCU state through the Serial-to-UART channel and then call the appropriate JTAG action with minimal effort and room for error. This also allows the user to monitor both aspects of communication between the PC and MCU in one central application, which is typically not the case for most IDEs. It is also possible to import the areas the user wants to read from or write to with a file, allowing the user to set up predefined memory maps and injection patterns rather than doing each operation manually.

Devices are automatically detected, and the user can toggle between selected devices. This allows them to test one board with fault injection while using another one as a comparison, have both boards subjected to fault injection at once if the physical setup allows for it, or simply just operate one MCU.

The MCU core registers can only be retrieved when the MCU is halted, so a shortcut was created to halt the core, retrieve the register values, restart the core and display the values to the user. The core itself can be halted, stepped, or run, and breakpoints can be set and cleared with a single click.

Reading data can be done either by specifying a start address and size, or importing a list of addresses to read from a file. Writing data cannot be done in blocks, and each 32-bit register must be written individually either by specifying the start address and data or importing a list from a file. A screenshot of the GUI is provided in Figure 4-3.

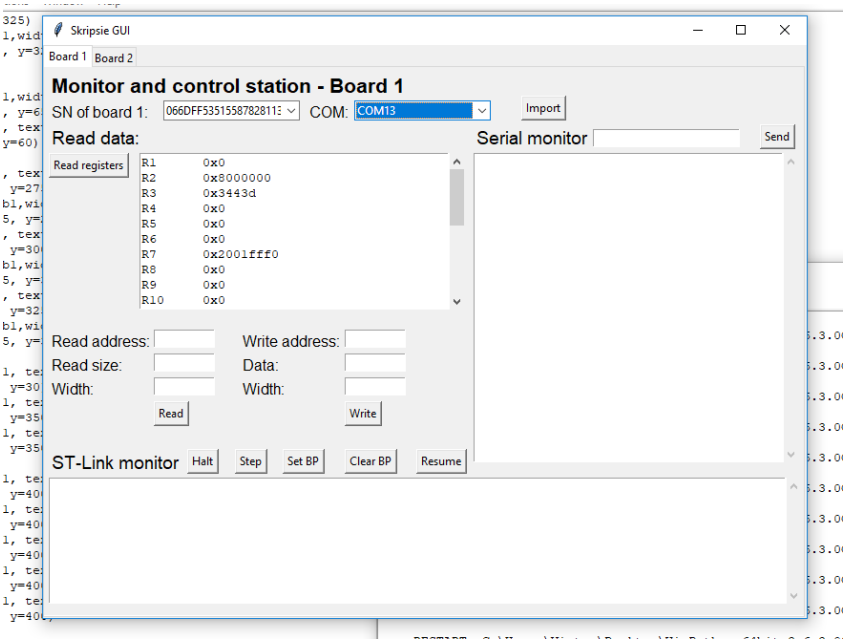


Figure 4-3 POC GUI

4.4 Testing the POC

The core focus to test for with the POC is functionality and speed. In this case, functionality refers to the system's ability to mimic the ST-Link CLI abilities and those of traditional IDEs to verify if JTAG can be used to extract and inject data, and speed refers to the system's execution time while performing data extractions. Other features, such as the UART communication, can be built upon in further iterations.

4.4.1 Functionality Testing

As the POC directly uses the ST-Link CLI, it should be able to perform all the operations of the ST-Link CLI. In turn, the ST-Link CLI is supposed to be able to perform its full range of features such as reading or writing to the DUT, halting, resuming or stepping the core, and uploading new firmware.

To test the ST-Link CLI the test MCU was connected to the PC, and every command available from the ST-Link CLI was manually entered and the output inspected. Any operation that would lead to a change in state of the DUT, such as halting the core or injecting data, was verified against a dedicated IDE, namely TrueStudio, to verify the operations were performing as expected.

Then the POC implementation was activated, and every available operation from the ST-Link CLI was rerun by the POC, this time comparing the output to that of the ST-Link CLI and TrueStudio.

To test the read and write abilities certain areas were read, some data changed and written back to the MCU, and read again to verify the changes. This process was repeated between the developed solution, ST-Link CLI and TrueStudio.

4.4.2 Speed Testing

When injecting errors to the DUT, speed is not critical, as the core can be halted and resumed as needed to create a pseudo-real-time simulation. However, when extracting data, the extraction speed is very vital. If it takes too long to extract significant amounts of data from the DUT it will either be impractical in a real SEE test, or the amount of output will have to be reduced, possibly leading to a loss of information.

To test the POC data extraction speed, an area of the SRAM was selected and repeatedly read. By performing this action enough, an average time can be determined to read a chuck of memory and the variance in execution times.

4.5 Analysis of the tests

This section analyses the tests performed in the previous section.

4.5.1 Functionality analysis

Comparing the custom library against the dedicated STM software shows that the developed library includes the full ST-Link CLI functionality. This is logical as it uses the entire ST-Link CLI as is. It is also seen that all of the ST-Link CLI operations work on the selected MCU.

When compared with the functionality of compatible IDEs, such as TrueStudio, it is noticeable that certain IDE abilities are superior, such as setting breakpoints and keeping track of program execution while stepping the core. However, this is not as important as reading and writing data via the ST-Link programmer. The developed solution adds functionality by allowing multiple boards to be connected and allowing the serial COM ports to be monitored, which cannot be done in the ST-Link CLI or TrueStudio. The developed solution could also read and write everywhere the dedicated software could.

It was also noted that uninitialised peripherals could not be permanently written to, regardless of using the POC, the ST-Link CLI or TrueStudio. The DUT documentation does not mention this, but it is speculated that to keep the device as power-efficient as possible, peripherals not in use are switched off and are not powered. The registers can be read, and the state can be changed via JTAG, but the moment the JTAG TAP releases control back to the DUT after the write it resets to a low. For this reason, all peripherals that are of interest need to be initialised, especially if their state is volatile.

4.5.2 Speed analysis

In terms of speed, the developed solution is overall faster to use than the ST-Link CLI, as it cuts the need for the user to call the ST-Link CLI and type every command manually. Still, it takes longer to execute if only the time to complete the commands is considered, thus excluding the time it would have taken a user to type the command. This is because the developed library is only a wrapper for using St-Link CLI, and every time the developed solution needs to communicate with the DUT via JTAG it first needs to initialise the ST-Link CLI, wait for it to finish executing and close, then finish its own execution. This does produce a lot of overhead.

Further, due to the way ST-Link CLI works a maximum of 19 commands can be executed per application call. This means that if 20 bits need to be injected into the DUT, it will take two calls to the ST-Link CLI. Thus, there will be double the time spent on initialising, discovering the board, connecting to it, closing the connection down again, and closing ST-Link CLI itself. This produces even more overhead.

When measuring the execution speed for extracting a 1kB chunk of memory, the average allocated memory per peripheral, it also revealed that speed is depended on whether it was extracted in the form of bytes, half-words, or words. 32-bit words took an average of 429ms to extract the 1kB chunk of memory and are 29.69% faster to read than half-words and 59.8% faster than bytes. Intuitively this does make sense as it would take four byte-sized reads through JTAG to extract the same amount of data as one word-sized read, and the MCU itself uses a 32-bit architecture, which requires extra cycles to process half-word and byte operations. These measurements are shown in Figure 4-4.

As shown in Figure 4-4, there is a large variance in execution time for the same actions, making it hard to track when the command is executed accurately. The time for Byte reads vary from 557ms to 1093ms, half-words vary from 800ms to 437ms, and 32-bit words vary from 809ms to 322ms. This variance is mostly because the PC does not prioritise the console command every time it is called. This variance can be almost double the fastest execution

time and will be inherent to using the ST-Link CLI. The variance also appears to be random, regardless of what else the PC is executing.

The data is sorted from high, the longest execution time, to low, the shortest execution time, as that makes the variance more visible. In practice, there appears to be no constant pattern to when operations take longer to execute, even when taking 10^4 samples instead of 100, and if the PC is executing complex tasks, like video rendering or idling in the background, it also does not make a visible difference.

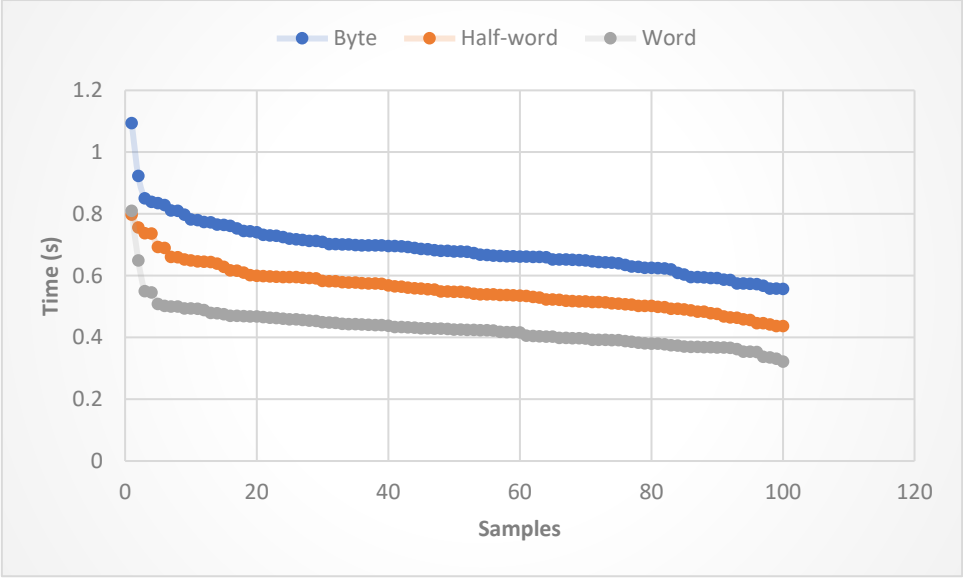


Figure 4-4 1kB extraction speeds, sorted high to low

This data read for this test is also a 1kB chunk of memory consisting of adjacent memory addresses, for example, from 0x00000000 to 0x00000399. It can become a much slower and more tedious process for peripherals as the needed registers are not adjacent memory addresses in the memory map. As such, a single call needs to be made for every non-adjacent register, no matter how small it is. All of these calls will also need to be mapped, as calls to unallocated memory addresses return an exception. This quickly becomes a low-level effort of finding which specific memory addresses are available to the specific DUT and might need to change for other MCUs.

For the chosen MCU it takes roughly 10 minutes (616s) to extract every readable bit from the processor. This means that 90-minute exposures in a radiation environment will need to be done to limit the read time to 10% of the total test time. For 8-hour testing slots, this is simply too slow.

4.5 The conclusion from the POC

The developed POC solution proves that it is possible to read data from and inject data to the DUT. However, on its own, it is not suited for SEE testing as it simply takes too long to extract all information. The timing can also vary depending on what the PC and its operating system are busy doing and prioritising. This is due to the ST-Link CLI that needs to be initialised for every call and then needs to detect and connect to the correct DUT. This relies heavily on the operating system, which might not prioritise it.

When combined with onboard fault detection to only extract data from defective areas, it becomes a more viable option. Still, as a result, any faults the DUT does not detect or that causes it to become completely unresponsive will be missed as mass data extraction is tedious both due to the effort of mapping all available registers and the time of extraction.

For fault injection, the speed issues do not matter as much, as the device itself can be halted and unaware of the injections. Since the injection process can be done anywhere, there is much less of a time constraint for overall execution speeds.

This approach's advantages are that it can interface with any MCU that supports ST-Link, which is a substantial number of devices and gives designers more flexibility to choose an MCU that perfectly fits their needs. It can also function as a stand-alone program or interface with the DUT directly via serial communication. It gives complete control over the DUT and can execute halts, reboots, and firmware uploads if needed.

4.6 Improvements for the next iteration

The POC shows potential, but it needs some alterations to be effective in a SEE testing environment. There are two significant concerns with the POC implemented solution. Its execution speed is slow, limiting testing flexibility, and it needs to be connected directly to the user PC, which cannot happen in a radiation test environment.

It is also tedious to hunt for specific memory addresses that are valid to read from the DUT documentation and quickly become very device-specific. Using the DUT to detect defective areas can quickly become very low-level and device-specific and make it hard to test real-life flight mock-ups. Execution speed is also very system-dependent and time-varying, which could lead to sizeable cumulative timing inconsistencies.

Chapter 5
Iteration 2: Getting test-ready

Chapter 3 explored the design's various aspects, and in Chapter 4 the core functionality was implemented. It was possible to read and write to the DUT, meaning that data extraction and injection is possible. However, several problems presented themselves and more functionality is needed before the system can be practically used.

This chapter describes the process of converting the POC to a system that is ready for a real SEE test at iTL. After the improvements, the system is tested, and the usability is re-evaluated to further mature the system.

5.1 Description of Iteration

The POC discussed in the previous chapter required the DUT to connect to the tester's PC with a USB cable. USB cables cannot be longer than a few meters, and since the tester cannot go into the radioactive environment found inside the testing vault, a dedicated testing station is needed inside the vault that connects to the DUT through the USB port to still have access to the ST-Link programmer and JTAG interface, while allowing the tester to be able to monitor and control the testing from outside the vault.

From Chapter 2.4, it is known that there is the option of ethernet communication to and from the vault. There is also a shielded area to protect monitoring devices inside the vault. To still use both the USB-to-UART communication and connect to the ST-Link programmer and connect to the tester PC outside the vault, a Raspberry Pi was selected to serve as a test station inside the vault.

The Raspberry Pi has ethernet and multiple USB ports allowing connection with multiple devices, including storage devices for logging purposes. It also has GPIO pins, which can be used to interface with hardware protection circuits between the DUT and the Pi, such as SEL detection circuits.

This also allows the use of Python on the Pi as it comes pre-installed. Also active by default is SSH, which can be used to communicate over networks and move files between connected devices, allowing for quick extraction of log files and easy uploads of DUT firmware of program changes remotely.

The POC also struggled with execution speeds as it was using an entire third-party program just for sending singular commands. The system's execution speed might be drastically improved by replacing the ST-Link CLI with a dedicated custom driver.

This iteration will be focused on including a dedicated test station in the system and reducing the execution time of JTAG operations by implementing a custom driver to replace the ST-Link CLI. After these additions, the system is tested at iTL to gauge its effectiveness in a proper test environment. A system diagram is shown in Figure 5-1.

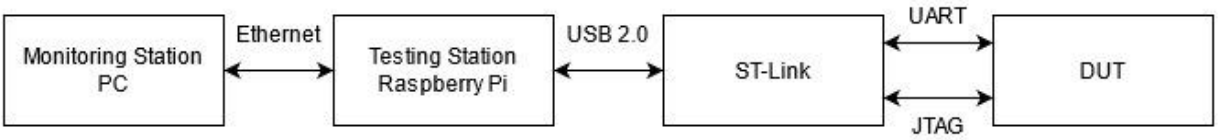


Figure 5-1 System Diagram for Iteration 2

5.2 Custom JTAG Driver

As seen in Chapter 4, the POC showed that reading and writing to the DUT memory and peripherals is possible but that it is not test-ready. With the POC, the ST-Link CLI was used as a device driver to communicate to the DUT. This led to execution speed problems because of the way the ST-Link CLI was called and used. It would be much more effective to be able to search for the DUT, connect to it and then stay connected for the duration of the test. A lot of the overhead is then mitigated simply by not opening and closing external programs every time a command is sent.

A different and more efficient driver is needed to connect the software to the DUT to do this. As this research was being done, multiple groups led attempts at open-source libusb-based, Python-driven drivers for the STM32 family [67, 68]. Their approach was to use a fast, effective C library called libusb that provides generic, user-mode, version-agnostic access to USB devices to aid in automated code flashing and debugging [66]. Libusb is extremely popular and typically ships with Linux and can be installed on most other operating platforms, including Windows. Due to its popularity, there are interface libraries available

in most programming languages, including Python, through a library called PyUSB. With the libusb and PyUSB combination, arbitrary data can be sent to any USB device.

Unfortunately, both teams did not have complete drivers and were still under extensive development at this research time. Therefore, as described in Chapter 5.3, additional headers were added to the Protoboard connected to the Pi so that a bare-bones version of the driver could be developed to suit this project's needs.

5.2.1 Implementation

The ST-Link driver protocol was partially reverse-engineered by monitoring the DUT and ST-Link CLI communication over the USB data lines with an oscilloscope to capture the raw data being transferred. Because USB communication is either at 5V or ground the oscilloscope trigger was set up to capture a snapshot of the data if it passed 4V. 80% of the full voltage was chosen to both catch signals not completely at 5V and to not falsely trigger on system noise on the data lines. This snapshot was then saved to the PC as an image, and the message data visually extracted. By sending different commands through the ST-Link CLI and monitoring the physical data sent, most of the standard command values could be found. The setup used is shown in Figure 5-2.

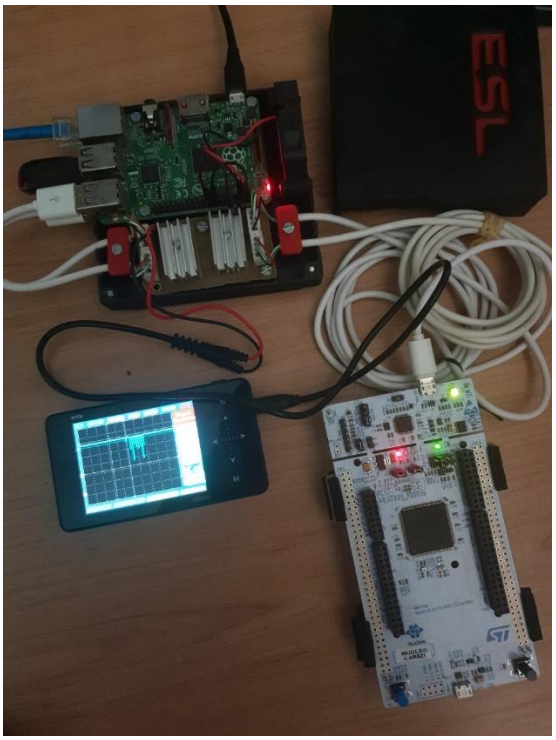


Figure 5-2 Using an oscilloscope to decipher the communication patterns between the ST-Link driver and the DUT.

This did lead to a loss of functionality, as not all communication data was deciphered. Only the minimum was implemented to get the application working, with the idea that this driver could be switched out for the ones developed by [67] or [68] once they are completed.

5.2.2 Analysis

The bare-bones driver could only connect in hot-plug mode with the frequency of 4MHz as that is the max the chosen MCU can operate at. It can read and write 32-bit registers, halt and resume the core and read and change the core registers. It cannot set or clear breakpoints, retrieve the ST-Link programmer serial number or connect in any other mode. Practically it can also not reliably work when connected to more than one DUT at a time due to the lack of a serial number as the identifier.

This was, however, enough to test one DUT at a time in the particle radiation vault, and the average read speed, tested in the same manner as in Chapter 4 by repeatedly reading a 1kB memory chunk went down from 429ms to 113ms, which is a 3.8x reduction in individual command execution speed. Figure 5-3 shows the comparison between the custom driver and the POC driver implementations when reading 32-bit words. The variance seen in the POC implementation is still relevant, with the custom driver taking between 84.7ms and 213ms.

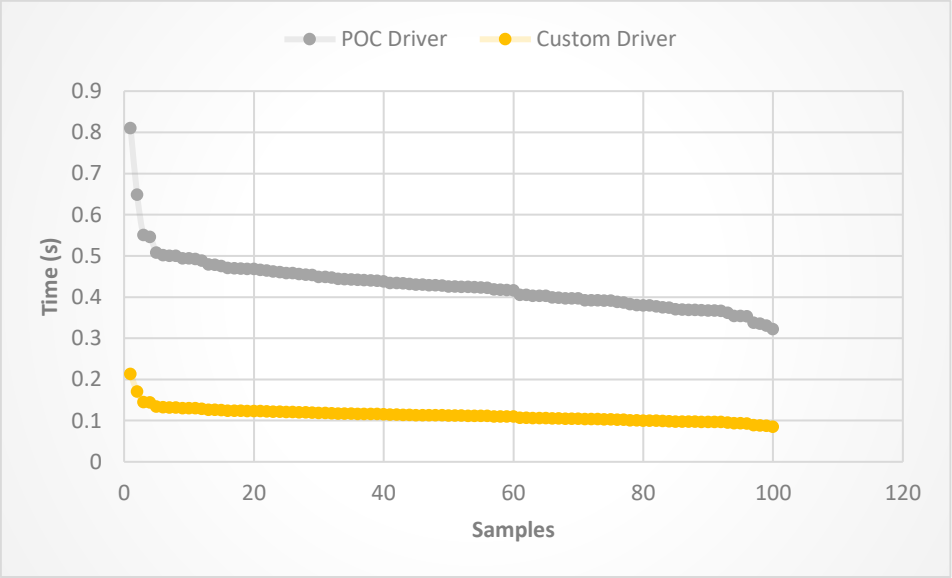


Figure 5-3 Driver comparisons of 1kB extraction speeds, sorted high to low

The time to extract all data from the chosen DUT is down to 72s, compared to the 616s. This means the new driver can read all data from the device 8.55 times faster and is much more manageable for real-life radiation testing. As expected, not having the overhead that comes with using the ST-Link utility directly impacts the speed performance. To limit the overhead to 10% when reading all the data in the DUT now only requires exposure windows

of 720s, or 12 minutes. This is remarkably faster than that of the POC, which would have required 90-minute exposures for a similar overhead percentage.

One problem with the bare-bones driver is that it is not completely compatible with all MCUs when it comes to writing to the Flash memory while the MCU is operational. The chosen DUT, the STM32L452RE, needs to unlock the Flash memory before any writing to that area can be done, which for this MCU is possible and can be done by writing specific values in order into the FLASH_OPTKEYR register [69]. This might be different for other MCUs, even if they are from the same brand. However, all read and write operations to SRAM were tested on and work on both the STM32F411E and STM32F334R8. As all three MCUs use the same ST-Link debugger and the same JTAG clock speed, testing the data extraction speed revealed almost identical results for all three MCUs.

Conclusion: The custom driver is a bare-bones implementation that can read and write data to the selected MCU much faster than the previous POC implementation. It also shows potential for near direct plug-and-play functionality for other STM32 MCUs that use the ST-Link debugger.

5.3 Test Station Design

The Raspberry Pi needs to switch the DUT on or off, monitor the serial communications if needed, and connect with the ST-Link programmer on the development board and the user on the outside of the vault via ethernet. A block diagram of the test station is given in Figure 5-4.

Power is supplied by a standard Raspberry Pi charger, the 5V fan is connected to 5V and GND on the GPIO, USB storage consists of standard memory sticks connected to the Pi's USB ports, and the data lines and 5V go directly to the DUTs from the USB connector via a standard USB cable. The switches are connected to the GND line of this cable to control current flow to the DUTs through GPIO pins, allowing the operator to remotely switch the DUTs on or off. Both JTAG and serial communication is transmitted via the data lines on the USB cables.

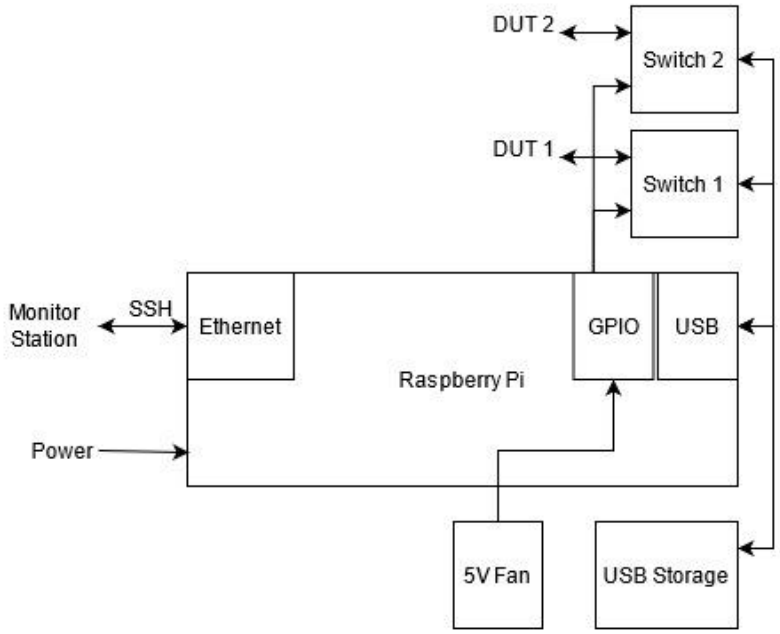


Figure 5-4 Block Diagram of Iteration 2 Test station

As the DUT was not going to be executing code for these tests, as noted in Chapter 5.1, the UART communication was not enabled because the DUT will not be able to use it. The focus is on the efficiency of the JTAG and the system overall.

5.3.1 Hardware protection

Certain SEEs have destructive effects, and one commonly found in particle radiation testing is SEL. To restore cells struck by SEL, power must ultimately be removed from those cells. As it is time-sensitive before damage occurs and hard to pinpoint where the SEL occurred, the entire device is typically power-cycled, and no information is captured regarding the whereabouts of the SEL in the DUT.

For the test performed in Chapter 5.5, the particle energy was 66 MeV with the option to degrade it lower, and is below 180 MeV, where SEL usually starts occurring. As such, no SEL protection circuits were added.

There is still be a need to power-cycle the DUT to prevent other effects, such as TID which is also happening in SEE testing, and it would be beneficial if it is off while not directly in the proton beam to avoid unnecessary SEE that could damage it, so hardware has been added to switch the DUT on or off.

This was done simply by connecting MOSFET switches on the ground line between the Raspberry Pi and the DUTs. The switches are controlled from the Pi's GPIO pins, which can be controlled from the software. A schematic of a single switch is given in Figure 5-5. This switch is implemented on the ground line of the USB cable connecting the Pi to the DUT

For this project, the IRF510 Power MOSFET was used as it was readily available and has a very low resistance, thus having a small effect on the overall operation of the DUT. The MOSFET reliability has not yet been tested in a SEE testing environment, but as it is shielded due to the setup, this should not pose problems. For the chosen DUT the voltage drop across the transistor was a minimal 18.1mV. On the DUT development board the 5V supply is regulated down to 3.3V before being supplied to the DUT. This small drop in supply voltage to the 3.3V regulator will not impact the performance of the DUT.

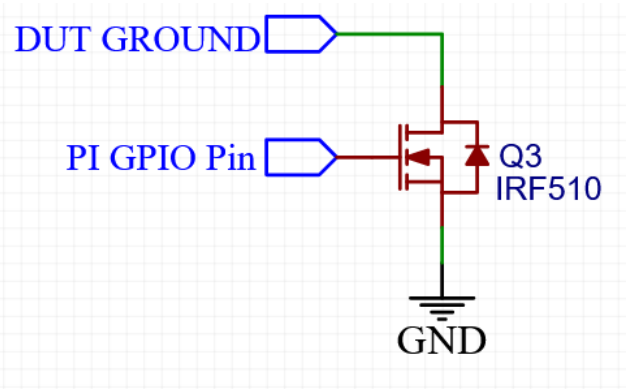


Figure 5-5 Single Switch Schematic

A protective case for the Pi and the switches was designed and 3D-printed, as that makes it more rugged for travel and handling and allows all wires to be clamped in place, avoiding bad connections caused by damage to solder points while the test setup was being done. A 5V fan was also added to aid in air circulation and cooling.

Connection headers were also added to the USB wires to allow monitoring with external probes. This is not a protective measure but is used for development for the driver in the sections above. The schematic for the entire case is shown below with a single DUT switch and a picture of the finished version, including two switches.

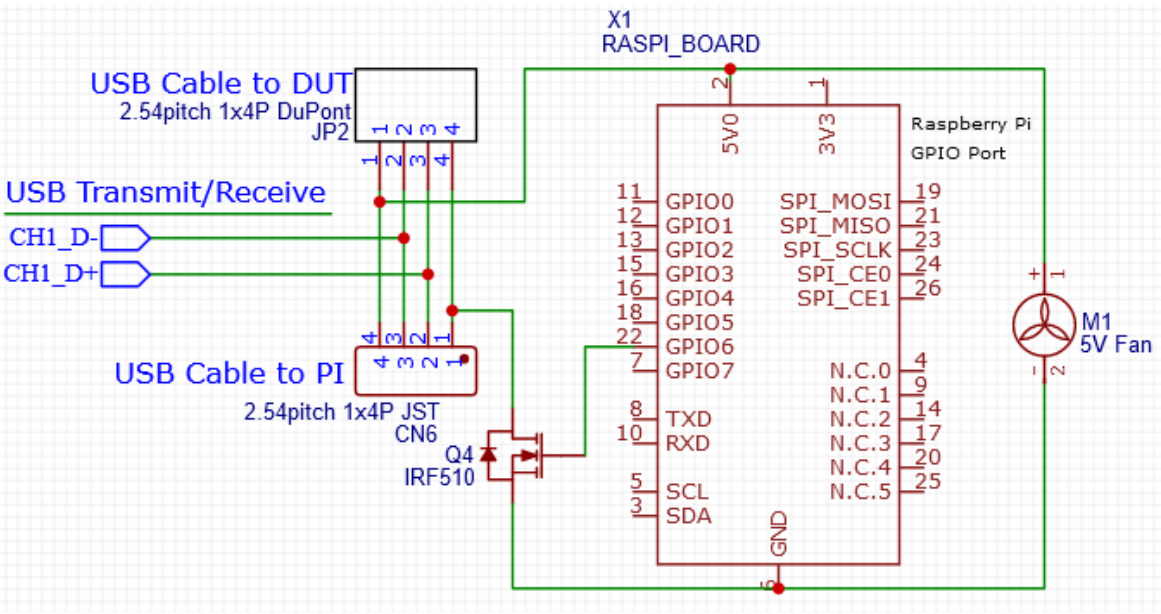


Figure 5-6 Test Station schematic with a single switch.

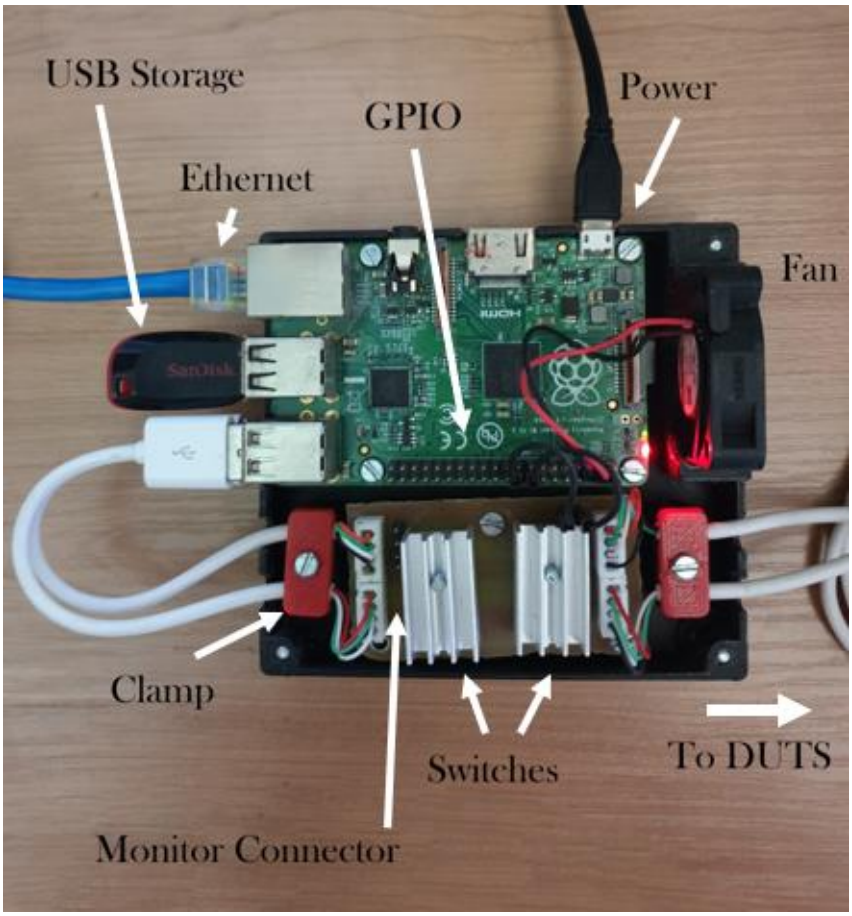


Figure 5-7 Picture of Test Station

5.3.2 Ethernet Connection

By using a Raspberry Pi and connecting it to the ethernet network leading outside the vault it is possible to gain access to the Pi's input and output streams via SSH. This lets a user log into the Pi and operate it remotely as if the input were entered directly into a terminal shell on the Pi itself.

This allows for a design where a Python script can be started on the Pi that implements a state machine, waiting for and reacting to user input entered on the other side of the SSH connection, creating an easy interface to communicate with the Pi.

This also allows the Pi's monitoring software to be completely detached in design to any ethernet communication protocols. It merely takes standard input as input, and print statements get sent to the monitoring station.

5.3.3 Software

The software on the Pi was designed to be modular so that individual parts of the program, especially the DUT drivers, can be completely replaced. This should make it easier to adapt the program for many testing environments and different MCUs. The Python modules are shown in Figure 5-8, and the final APIs at the end of this project is given in Appendix B.

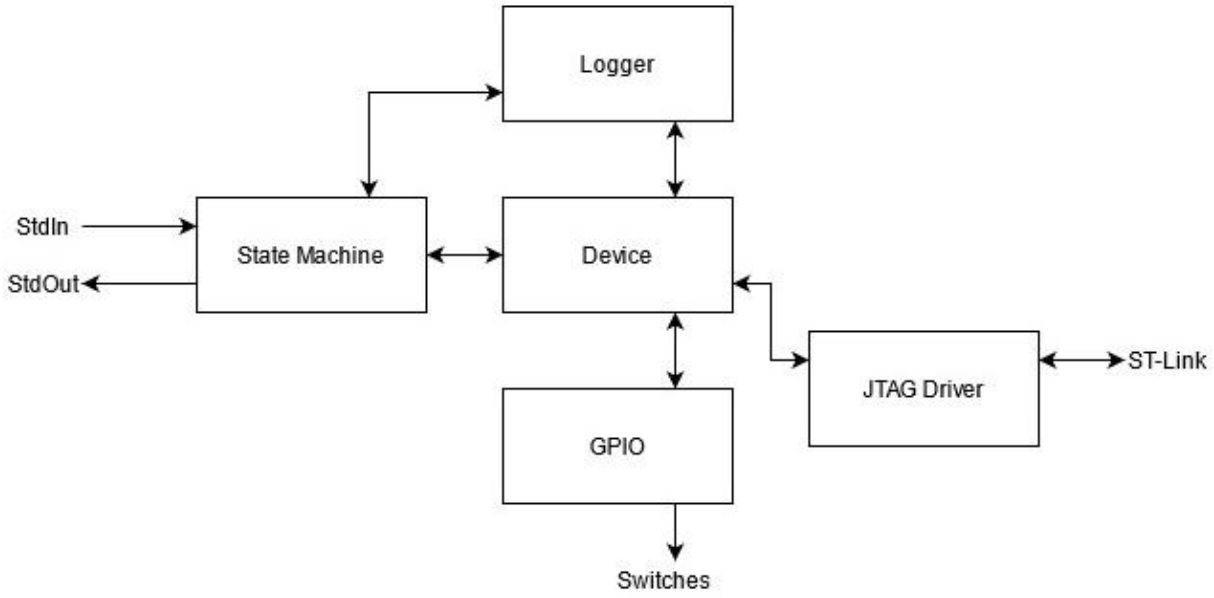


Figure 5-8 Modular Python modules used in Iteration 2

5.3.3.1 State Machine

Commands received from the Standard Input are put in a queue and then executed from oldest to newest, implementing a FIFO approach. A command has a specific protocol that determines which action is being executed, and when received the program executes the command and, upon completion, processes the next command.

Each DUT is represented by a Device class, which splits into a GPIO Driver instance and a JTAG Driver instance. This makes it easy to simply change the class definition to operate different devices. The commands are sent to the respective instances where they are executed, log their results to files if needed, and then print out any feedback into the Standard Output where the Monitoring System can interpret it from outside the vault.

A global Logger class manages writing to files and lets the monitoring system know when to download the log files to get backed up outside the vault to minimise the chance of corruption. All files are written to three locations locally. One version in the folder the application is running in, and two on separate flash drives.

Commands are space-delimited strings, with the first value being the type of command and the next the specific parameters required by that function. They are summarised in Table 5-1:

Table 5-1 Communication protocol between the test station and the monitoring station

Cmd	Description
01	Toggle Alive: If active, every second the DUT is checked if it is still responding.
02	Set File 1 Location: Set the 1st location for the log file to be stored.
03	Get File 1 Location: Retrieve the 1st location where the log file is stored.
04	Set File 2 Location: Set the 2nd location for the log file to be stored.
05	Get File 2 Location: Retrieve the 2nd location where the log file is stored.
10	Switch 1 ON: Set the 1st GPIO instance to switch on.
11	Switch 1 OFF: Set the 1st GPIO instance to switch off.
12	Switch 2 ON: Set the 2nd GPIO instance to switch on.
13	Switch 2 OFF: Set the 2nd GPIO instance to switch off.
20	Full Write On: Write pattern everywhere
21	Full Write Off: Write pattern only in selected blocks. Parameters: Selected memory blocks
22	Set Pattern 00: Set the pattern to write to 0x00
23	Set Pattern 00: Set the pattern to write to 0x55
24	Set Pattern 00: Set the pattern to write to 0xAA
25	Set Pattern 00: Set the pattern to write to 0xFF
26	Write Pattern: Writes the selected pattern to the DUT.

	Parameters: DUT number
27	Write Memory: Write data to a specific memory address. Parameters: DUT number, memory address, data
30	Read Memory: Reads a data block from specific memory addresses. Parameters: DUT number, memory address, size
40	Start Core: Starts the selected DUT core. Parameters: DUT number
41	Halt Core: Halts the selected DUT core. Parameters: DUT number
42	Step Core: Steps the selected DUT core. Parameters: DUT number

5.3.3.2 GPIO Driver

To control the GPIO pins for DUT power control, the RPi.GPIO module, by default installed on recent versions of Raspbian Linux, is used. A class was created that can initialise a pin, set its state as either high or low, and reset the GPIO to its safe defaults if the application ends. Thus, every pin that is needed can get an instance of this class, and each instance is then linked to a DUT device so that it can easily be controlled from the state machine.

5.4 Monitoring Station

The monitoring station allows the tester to connect to and monitor the system inside the vault. It needs to be on the same ethernet network as the test station and establishes the SSH connection to establish communication with the Python state-machine on the Test Station.

As the software on the Pi can be directly controlled through SSH, no interface is needed if the PC can establish an SSH connection. This is built-in for most modern operating systems. Thus, the user can have direct control by starting an SSH session. However, as seen with the POC, it can be a nuisance to type every single command, and as such, an interface that makes the Test Station more user-friendly is implemented.

The Monitoring System is also implemented in Python. This software can establish an SSH connection to other devices and will also have a GUI to enable the user to select and implement commands effectively.

To manage both the SSH communication and the GUI each is run in separate threads; otherwise, the program continually enters a blocking state.

5.4.1 SSH Communication

To use Python to establish and maintain an SSH connection needed to communicate with the software on the Test Station an external library is required. This library is Paramiko [70], a pure Python interface around SSH networking concepts implementing a large portion of the SSH feature set. By implementing Paramiko, it is easy to establish an SSH connection, through which data can then be sent to and received from the connected client. All it requires is the IP address and the username and password to log in.

Paramiko also makes it possible to execute Simple File Transfer Protocol (SFTP) commands, which is used to clone all three versions of the log files from the test station to the monitoring station, where it gets scanned and compared to each other as well as with the reference pattern described in Chapter 5.5.2 to count the amount of SEU.

5.4.2 Graphical User Interface

The GUI has a few significant areas on its frame. There is a connection segment where the SSH connection to the Test Station is established. Once established, any output from the Test Station appears in the terminal display, and any input entered into the terminal input is sent as-is via SSH. This allows the user to operate the Pi through a terminal interface if needed.

There is also an area selector. This toggles which memory addresses to perform pattern read/write operations on. It is configured by translating the memory map into a text file and allows nested groupings to be easier to use. This is handy if certain parts of the device need to be excluded from the read/write process.

The last block contains buttons that automatically compile and send the commands to the Test Station as defined in Table 5-1. A local logging system can be used to timestamp when the proton beam was activated and what the current used was. Unfortunately, the system has no way of knowing when the beam is active or not and needs to be tracked and synced by hand afterwards. The GUI is displayed in Figure 5-9.

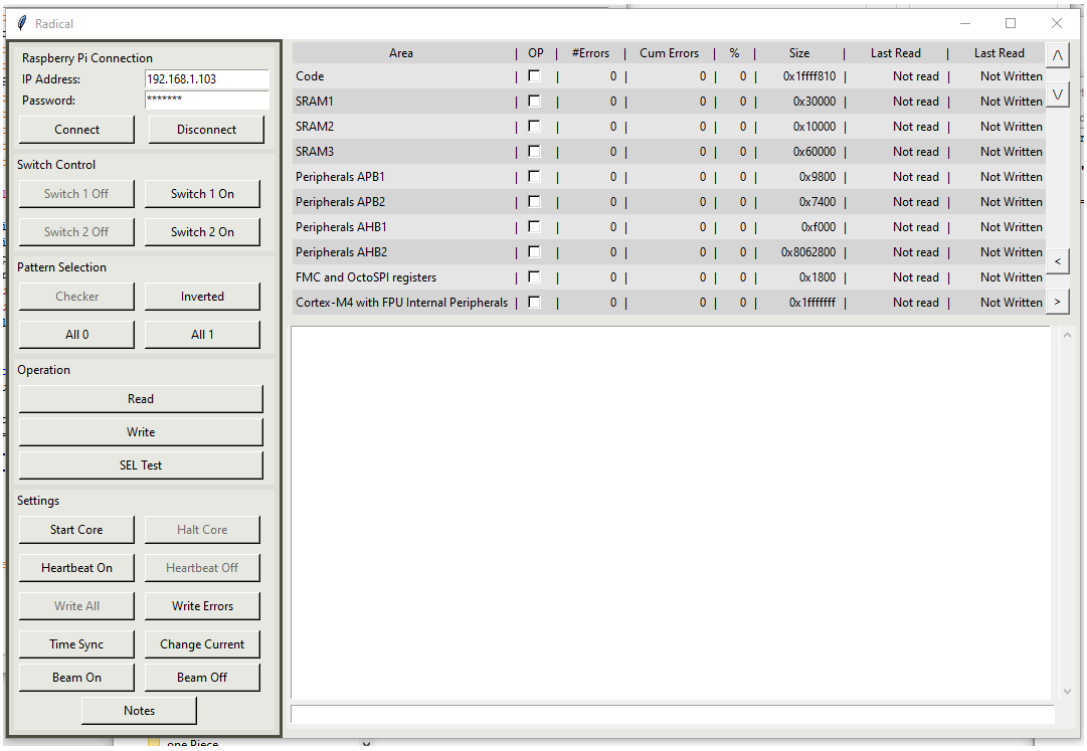


Figure 5-9 Updated Radical GUI

5.5 Testing at iTL

On 29 January 2020, this system was tested on the D-line at iTL, with the setup starting on 28 January 2020. This section describes the test goals, as well as the actual test process. An analysis of the entire iteration is done in Chapter 5.6.

5.5.1 Radiation Test Goals

The primary goal for this test is to determine the effectiveness of the current system in a real SEE environment. Any strengths can then be built upon, and system weaknesses can be identified and improved.

A secondary goal is to get SEU and SET sensitivity data on the selected MCU, the STM32L452RE. There is interest from a local company to possibly use this MCU in the future. Any information regarding its radiation sensitivity will be of aid to them.

SEU and SET sensitivity data can also be used to map the probability that individual bits will flip when exposed to the beam. Using this map out-of-beam to predict where SEU or SET will have caused an upset might be able to allow testers to simulate the effects of the proton beam for a certain particle LETs by manually injecting errors into these predicted

bits. This map can then be used to inject errors with the same probabilities found during SEE testing to view the effects on the DUT when actively executing code without using the proton beam. This will aid designers for future tests on this MCU by generating more realistic upsets, allowing them to more effectively mitigate the sensitive code sections, be even more efficient with beam-time at iTL, or possibly reduce the number of SEE tests that are needed altogether. This is discussed in more detail in Chapter 7.

5.5.2 Radiation Test Plan

For the test conducted at iTL, it was decided to attempt to get a baseline SEU sensitivity for the chosen DUT. Before the run, the memory map will be filled with different patterns. Four different patterns will be used to determine if there is a bias towards bits flipping high or low in the case of SEU. The patterns consist of writing 0x0 to all bytes to get all bits low, 0xff to all bytes to set all bits high, 0xAA (10101010_2) to all bytes to alternate between highs and lows, and 0x55 (01010101_2) to invert the previous pattern.

As noted in previous sections, certain areas cannot be written to if not initialised for this MCU, and not all device registers are adjacent memory addresses. Before the test, all the patterns will be written to the DUT, and the resulting pattern that can then be read back is used as a baseline. This baseline can then be compared with the DUT memory after a run to determine where they differ and thus an upset has happened. This accounts for the uninitialized areas and the necessary areas to initialise the DUT.

During the SEE test, but before a run, the memory map is filled with one of these patterns. The run is then started and the DUT is exposed for a certain amount of time, and then the pattern is checked for deviations to see if any bits have flipped. A new pattern is then written to the device, a check is done to see if any bits are stuck and did not write, and then the process is repeated.

The core is also halted to avoid program execution and clock signals to mitigate SETs. SETs typically only cause upsets in memory elements when arriving as an element's clock changes state, allowing the error to propagate. This will mean that the majority of errors will be SEU, and from this a base SEU error rate can be determined. If the error rate is too low, the next run can have either a longer exposure or increased beam current to increase the error rate to a statistically significant level, but with a small chance of overlapping SEU on the same bit.

When the baseline SEU rate is determined, the core can be resumed, and an increase in errors is expected due to SETs causing SEUs more frequently. The total number of SEUs can be deducted from this increased number of errors to determine the SET sensitivity.

Predicting how many SEU are inherently caused by SET might help satellite developers choose better mitigation techniques, as SETs appear to be inputs to the memory elements, rather than the transistor directly switching state inside the element, which can more easily be detected by safety features such as CRC.

After all the runs, a baseline sensitivity map, as mentioned in the previous section, can then be generated by comparing the baseline patterns with the extracted patterns during the test and placing a probability on cells to experience SEU or SET for a certain particle LET. Should certain areas of the DUT experience no SEU, while in other areas, many errors were induced, this map will reflect those probabilities of SEU and reveal which areas need the most mitigation should they be used.

5.5.3 The Test

The testing setup was done the day before the test was scheduled and followed the standard setup as described in Chapter 2.4. The tests were done on the D-line at iTL. The data room was used as the control room for these tests. The radiation test plan, as mentioned in the previous section, was followed during testing. Figure 5-10 shows the DUT in line with the beam, and Figure 5-11 shows the test station among the other electronics behind the lead shield.

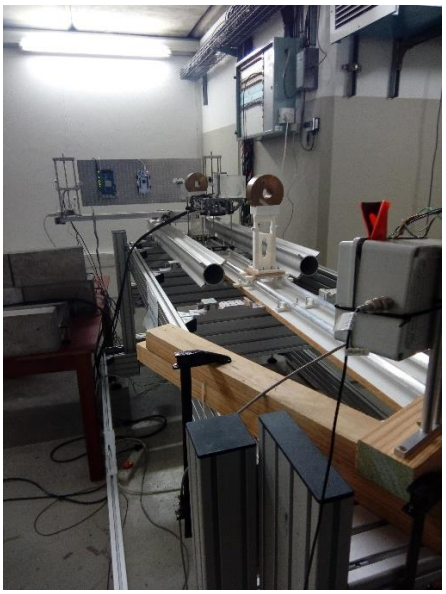


Figure 5-10 DUT aligned with the beam

Figure 5-11 Test station on a shielded table

During the test, the DUT was exposed six times for a total of 47 minutes. The first three exposures were at a beam current of 20nA and comprised of a 2, 5, and 10-minute run. In the first run, one suspected upset was detected. For all runs, the beam energy was at 66MeV, the highest that was available that day.

To try and force upsets, the beam current was raised. The rest of the exposures were at a beam current of 41nA each, which is the highest it could go for the vault configuration. Each exposure was 10 minutes long. No upsets were detected in the DUT during these runs. However, the ST-Link programmer, which was also relatively exposed to the beam, became unresponsive twice but became functional again both times after power cycling the DUT.

Fluence data for these tests can be found in Appendix A.

5.5.4 Testing Issues

The designed system struggled to communicate over the network and the connection was frequently dropped when using the GUI. Due to built-in features to prevent the user from clicking the wrong buttons, the DUTs could not be turned on in the GUI if the SSH connection is not established. This made most of the GUI redundant during this experiment. The code that extracted and analysed the data from the log files from the test station to the monitoring station was also locked due to this. However, a user-initiated SSH session directly from the command line could stay connected, so the testing was done without the GUI. The original analysis scripts, that compare the extracted data to the baseline maps, were called per-function in a Python interpreter, bypassing the GUI, and worked as intended.

The network problems only happened when connecting via Paramiko, so two command-line consoles were opened on the monitoring station and manually connected to the test station. One was connected with the Python interface on the test station and used to control the DUT, and the other one used SFTP to clone all the log files to the monitor station as the tests were done. The files were then compared using the original program to find upsets, as mentioned above.

Another concern was that virtually no upsets were being detected, with the core halted or not. Initially, the runs were extended, irradiating the DUT longer and giving greater exposure. As there were still virtually no upsets, the beam current was increased to the facility's maximum for the vault that was being used. After a cumulative 30 minutes of maximum exposure, it was decided to move to other tests as other DUTs and systems also needed to be tested during the test slot.

5.6 Analysis of Iteration 2

This iteration delivered remarkably faster data extraction rates by using a custom driver, and it should be easy to modify for many other STM32 MCUs. It can perform all the basic operations needed for this project, but it lacks more advanced features and is, all things considered, a very bare-bones driver. However, the speed increase meant that the data extraction time was satisfactory for a real SEE test.

The use of a Raspberry Pi as a test station was overall effective. It is readily available, affordable and its operating system, Rasbian Linux, comes with everything used for this iteration already pre-installed. There were problems on the test day between Paramiko and the Pi. It is suspected to be due to a misconfiguration on the Pi due to human error that caused it to identify the autonomous connection from Paramiko as insecure. This issue will be investigated in the next iteration.

The GUI was clunky and completely rendered useless when a connection could not be made to the Pi due to "safety locks" placed on the buttons that required the device to be connected, even when doing local activities such as comparing downloaded data files to find upsets. Luckily, the comparison functions could be called manually from the Python interpreter without using the GUI. The GUI could have been better implemented.

Unfortunately, during the test, only a single upset has been induced, in the SRAM during the very first run. This could be because the cross-section of the device might be too small for the exposure it experienced, and longer exposures were needed, or exposures with a higher current to increase the chance of SEU. It could also possibly be due to a failure in the design of the system that resulted in no detections. To verify, after the test, the raw, extracted data from the processor was re-evaluated to ensure that the system functioned as it was designed to. All data files were compared and identical to the pre-radiation versions, except during the first run. No data is generated by any part of the system other than the ST-Link programmer, so all data must have originated from there and, supposedly, from the DUT down the line. It was concluded that there was indeed only one upset detected on the DUT itself.

This test result does not give us much insight into the tolerance levels of the DUT, because there is simply not enough data to eliminate test errors. As a result, no system sensitivity map, as discussed in Chapter 7.1, can be calculated. However, the test has served its purpose by clearly highlighting some areas in the design that can be improved to provide for a more reliable testing experience, which was the primary goal.

Chapter 6
Iteration 3: The Improved Experience

This chapter further betters the system design by implementing fixes to the shortcomings experienced during the SEE test to improve the overall usability and user-experience.

6.1 Description of Iteration

In the POC, the designed system takes too long to extract all data from the DUT, limiting the effectiveness of the system and that it cannot be used at the testing facility at iTL because it would need a human operator in or very close to the vault. This cannot happen due to health risks to the operator.

However, the previous chapter introduced Iteration 2, the second iteration of the design, which made use of a custom driver that increased the system speed and allowed for remote testing through SSH, by using a Raspberry Pi as a test station inside the vault, connected to the user on a monitoring computer outside the vault.

Iteration 2 was then tested at iTL and barely met the test's needs, and it seriously lacked in user experience and was hard to use. The GUI was clunky, with the area to select peripherals to read to and write from not navigating quickly or intuitively and with many of the features locked due to a design oversight, forcing the user to access the backend code through the Python interpreter to be able to use some of the functionality.

The SSH connection that was supposed to be managed by the software, to let the user focus on other aspects of testing, completely failed, and the SSH terminal and file retrieval from the testing station had to be done manually.

Further, the expectation for the test was to gather a baseline sensitivity map for the DUT, namely a probability map that can be used to replicate the SEU pattern. This map could then be used to test various firmware applications to inspect the effect of the simulated radiation, in the form of injected errors, on the firmware's functionality to detect SEU-sensitive areas. Should the same firmware be used during a SEE test and similar results are achieved as with the injection tests, it would verify fault injection through JTAG as a viable way to simulate the effects of SEU on the system. As the test progressed, it became clear that virtually no SEUs are being detected and that generating such a probability map will not be possible due to a lack of statistical data.

To improve the design of Iteration 2 these aspects would need to be addressed. A feature that was also added is the ability to automatically execute scripts, opening the possibility for more automated testing. The lack of a SEU probability map is discussed in Chapter 7, which focuses exclusively on fault injection. An updated system diagram is shown in Figure 6-1.

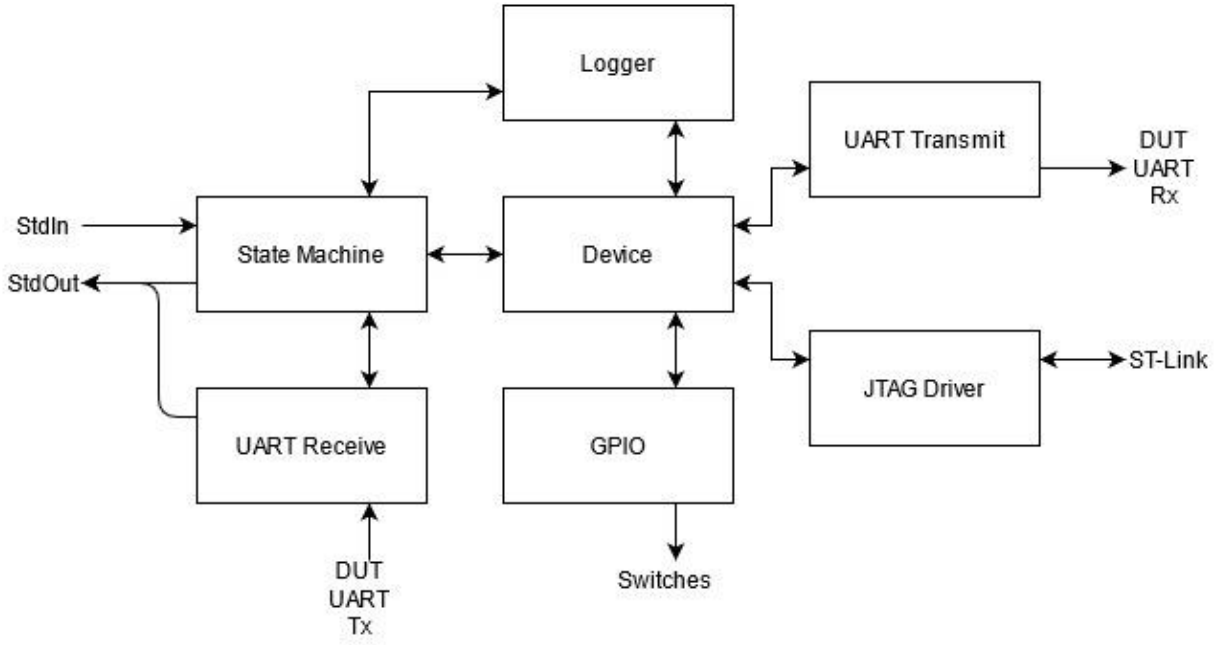


Figure 6-1 System Diagram for Iteration 3.

6.2 SSH Communication

As the access to the radiation vault is limited, reliable communication between the user and the DUT is key. The reason communication dropped between Paramiko, the Python library used to establish SSH communication, and the test station is unknown. It is thought to have

been a user error when originally setting up the Raspberry Pi, and fixed by reinstalling the operating system onto the Raspberry Pi with the default configuration. It was then tested on almost all of the original network the connection kept getting dropped on, and performed flawlessly. The only physical change between this setup and at iTL was the much longer ethernet cable used to link the vault and operations room.

The fault being gone was slightly problematic because, with the exact cause unknown, the problem might still be present and masked by an overlooked element in this verification test, so an inspection of how Paramiko was implemented was conducted.

Initially, the SSH channel was programmed to run in a separate thread to the GUI, theoretically allowing both threads to execute with minimal blocking between them. The main thread contained the GUI, and a secondary thread the SSH connection. Through a deeper inspection of how Python's threads are implemented and after searching for similar problems on the internet, it became apparent that Python's main thread will get preference over other threads. This can sometimes lead to the secondary threads being slightly blocked and can cause problems if the thread needs to be responsive for applications such as monitoring an input stream.

The slight blocking might be disturbing the SSH communication, and the much longer, possibly quite old, ethernet cable used at iTL might have aggravated the issue. With this in mind, the test setup was replicated on the same network it had been developed on, not the one configured at iTL. On this network, the problem has thus far never happened before. The only change was to replace the ethernet cable connecting the monitoring station to the hub the test station was connected to with a 50m long ethernet. This was done to try and mimic the long cable used at iTL to connect the control room and vault. This change is shown in Figure 6-2 in red.

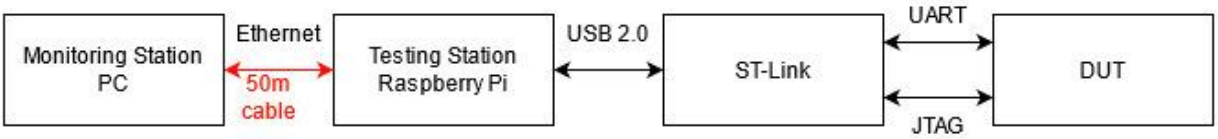


Figure 6-2 The system with a replaced, longer ethernet cable.

A connection was established through Iteration 2, as it was when testing at iTL but with the reconfigured Pi. Sure enough, the connection consistently dropped within three hours.

This is not as regularly as at iTL, but it is assumed the ethernet cable is much older and worn down than the one used in this test. As a comparison, a normal terminal console was also used to establish an SSH connection and could stay connected throughout an entire day before being terminated by the tester.

Since a normal SSH session stayed connected and the Python implementation consistently dropped, it is clear the problem is with the Python implementation. The SSH connection was moved to the main thread and the GUI to the secondary. A connection was then established to the test station using the Python interface after the change, as well as through the terminal console. None of the channels dropped once over three days.

It was then concluded that this was most likely the original cause, or at the very least, that the system is more reliable with this small change.

6.3 Graphical User Interface

To implement a better user experience, a more efficient GUI was needed. GUI-building in Python is very limited, and it was decided to convert the GUI from Python to Node.js, an environment that uses JavaScript as the core programming language. The rest of the project remains Python-based. Node.js is an extremely popular platform for web development as both server side and client-side scripting can be implemented in one language. The advantage to using Node.js is that the GUI can be built out of website components, made from HTML, CSS and JavaScript.

The methodology was to use Node.js to initialise a local web server, which any other device can then access on the network through a standard web browser. The GUI is made as a traditional website by using HTML for the element layout and CSS for the styling. The Electron module was then used to convert this into a standalone application. This has the added benefit of running as an executable file, but the assets can still be modified. This means an executable can be created containing both the Node.js environment and the Python environment, but all Python scripts and HTML, CSS and JavaScript files can be modified. This can now simply be copied to the monitoring station PC, without needing to install any environments.

This change made it incredibly easy to create a fluent, responsive interface with easily customisable buttons and functionality that is also very intuitive to use. As CSS is used to style individual elements, it was also possible to easily add features that are much more complex to implement in Python, such as marking certain parts in the console output in certain colours; for example, messages coloured red indicate errors or messages coloured yellow indicate warnings. A responsive design allows the screen to adapt to any size, even

on mobile devices or tablets. Since the SSH functionality is needed to communicate with the software on the test station, the Node.js web server interfaces with the Python script on the monitoring device to replace the Python-based GUI while still allowing all other, already implemented functionality.

The goal of this GUI is simply to replace the previous one, and all the options have remained. However, buttons and commands have been grouped, allowing certain safety features without affecting the other parts of the GUI. This enables checks to prevent an attempt to send commands meant for the Python script directly into the Pi's terminal. Incorrect commands directly into the terminal could possibly cause problems if the script failed to start or crashed due to an uncaught exception. This still allowed other functionality like attempting SMTP even if the original SSH connection failed or analysing the data extracted from a DUT even if no connections are active anymore. Previously, if one part of the system failed, everything would be locked. A section for the virtual COM port communications has also been added. A screenshot of part of the GUI, where the connection is established, can be seen in Figure 6-3.

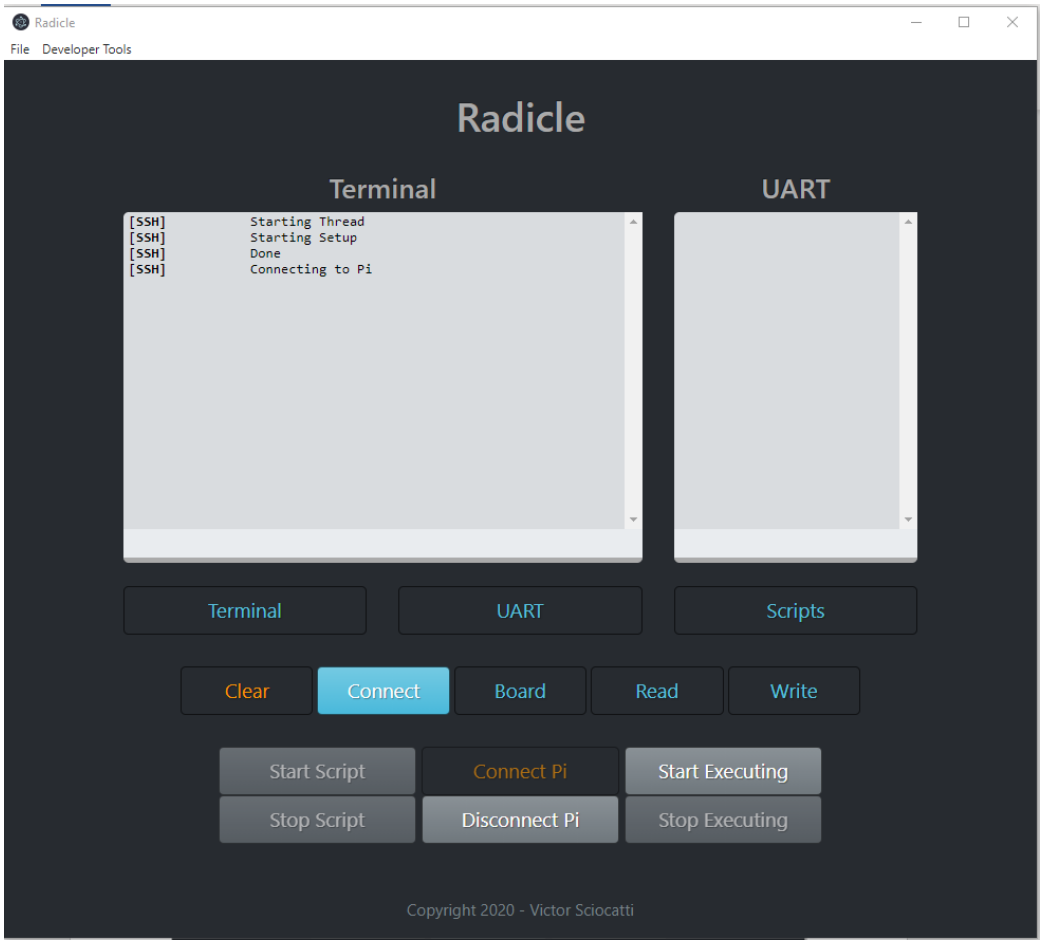


Figure 6-3 Final GUI snippet

6.4 Scripting

Up to this point, controlling the DUT and reading and writing data to and from the DUT via JTAG has mostly been done manually. Even when using files to define where to read from and write to, the entire file was processed in one go.

However, what if the user wants to run an initialisation procedure, such as switching on the DUT through the GPIO pins, connecting to the DUT, flashing new firmware, waiting for confirmation from the UART the DUT is ready to test, and then notify the user everything is good to go? Or if, following directly on the initialization, a new script starts executing procedures, sending commands over UART to navigate the DUT test procedure and doing JTAG reads every 3 minutes until the user aborts the process?

This can now be realized. In order to further improve the usability of the test suite, the ability to use scripts was added. There is a master script that is used to navigate between various smaller procedure execution scripts. The master script just links scripts sequentially or performs a jump to a previous script in the chain to create a loop.

The smaller scripts can be filled with any command the user could use to interact with the state machine on the test station and an option to delay between the commands by set amounts.

Thus, an initialization script can be made containing instructions to produce the example sequence above. A test script can be made for further procedures. A third script can be made to upload entirely new firmware, reinitialise the DUT and wait for user input before executing another script that repeatedly injects a set number of faults every second by halting the core, inserting the data, and resuming the core.

With this feature added to the project, it greatly expands its useability. Not just for simplifying operations during SEE testing, but also for fault injection, as discussed in the next chapter. An example script process, that performs the steps executed manually for the SEE test at iTL in Chapter 5.5 for two patterns is depicted in Figure 6-4 below. The grey blocks are the elements in the master script, and link to their respective secondary scripts. Each white block is a step in these secondary scripts. The master script can only progress to a new secondary script if the current secondary script reaches its last stage and is released. At any point, should the need be there, can the scripting process be halted.

Running the master script in Figure 6-4 will automatically connect to the test station over SSH, start the relevant scripts, power on and connect to the DUT JTAG and virtual COM port interface, write 0x0 to all memory addresses, extract those memory values and verify

that they did write correctly. At this stage the DUT is ready to be irradiated, and waiting for user input, either by clicking “Next” or “Abort”. The DUT can then be irradiated, and immediately after the run the user can click “Next” to automatically extract all data, compare and search for upsets, and move on to the next script.

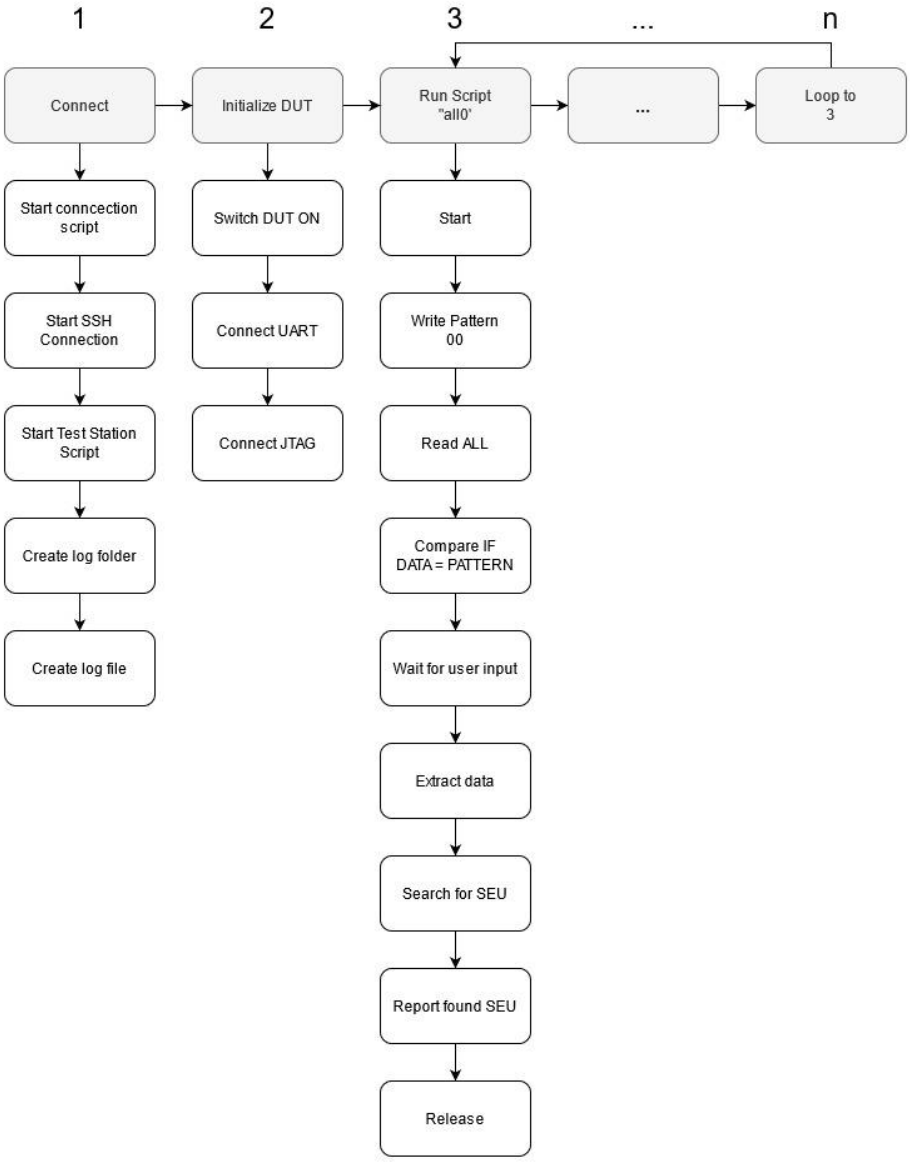


Figure 6-4 A depiction of the scripting progress

6.5 Analysis of Iteration 3

The reworked user interface appears much smoother to operate and is easy to use. It does come at the cost of an extra environment, but it can easily be reverted back to a Python-

based GUI should the need be there. Having the GUI outside of Python also helps to give Paramiko and the SSH implementation more time to execute without being blocked, leading to more stable network connections.

The implementation of automatic scripting is valuable for testing devices while the beam is not accessible, or for preparing for SEE tests by verifying the implemented mitigation results.

The main focus of this iteration was user experience. The biggest problem of this iteration is that it is untested in a SEE testing environment, as no tests could be scheduled at iTL due to safety concerns regarding the Covid-19 pandemic. Thus, before conclusions can be made about whether the GUI and scripting works in a physical radiation test, a physical radiation test will need to be performed.

Chapter 7
Fault Injection

While the development of the test suite itself mainly focused on effective ways to get data out of MCUs, JTAG has made it relatively simple to get data in by manipulating actual memory cells and peripherals. This will most likely not be used during a real SEE test, but can be handy to predict system responses to SEU and verify the effectiveness of mitigation techniques.

This chapter focusses on fault injection through the developed system and possible use-cases of fault injection to verify mitigation techniques before SEE tests or to find system vulnerabilities.

7.1 Beam simulation

If the execution loop of the DUT is kept extremely small, much of the Flash memory and SRAM and all the peripherals are active, but not in use. It was hoped to develop a method that can determine an MCU baseline SEU sensitivity by initializing the MCU, and then halting the core and writing a set pattern to these unused bits. This way, during irradiation, most of the DUT can be monitored to see where SEU occurs. After irradiation, the pattern can be retrieved from the DUT. Any differences from the original pattern can be classified as an upset.

Upsets should be mostly SEU and not SET due to the clocks being halted. SEL should not appear if the energy is low enough, namely below 180MeV/particle. After enough SEUs were detected to create a probabilistic model of the device's SEU sensitivity, the core can be resumed, and the test repeated. Hypothetically, the number of errors detected will increase and be the sum of SEU and SET. The difference between these two results will be the effect of the SETs.

The probability of any bit experiencing an SEU or SET can then be mapped on to the DUT memory map. This new map can then be used out of the beam to inject errors into the DUT in the same distribution as seen during SEE testing, thus simulating the SEU and SET effects the beam would induce. Developers could then use this to get a feel for how their systems will react when performing SEE tests, without having to continually retest after every significant change.

The DUT can be tested once to get the distribution map, and then the final implementation of the DUT, functional and mitigated as it would be in-orbit, can be tested. This could reduce in-beam testing needs, especially if a database of these distribution maps could be generated for various MCUs. Developers will only have to test their implementation on the MCU, saving cost and development time.

Unfortunately, not enough data was generated during the test described in Chapter 5.5 to be able to create such a distribution map, and further testing could not be achieved due to COVID-19 safety regulations that were put in place shortly after the first test.

At this moment, this approach is only hypothetical, and a simple version of this approach is suggested here, and can possibly be further investigated in future research. Due to the lack of test data, an SEU upset rate is assumed for a hypothetical DUT that only consists of eight bytes of SRAM, as that is easy to visualise. This hypothetical DUT has been vigorously exposed over R runs of M minutes each with a fixed LET. The total upsets per bit is represented in Table 7-1, denoted as Z_{iBitj}, where i is the byte number, and j the number of the bit inside that byte.

Table 7-1 Visualization of upsets across eight bytes of SRAM cells.

	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
Byte 1	Z_{1Bit0}	Z_{1Bit1}	Z_{1Bit2}	Z_{1Bit3}	Z_{1Bit4}	Z_{1Bit5}	Z_{1Bit6}	Z_{1Bit7}
Byte 2	Z_{2Bit0}	Z_{2Bit1}	Z_{2Bit2}	Z_{2Bit3}	Z_{2Bit4}	Z_{2Bit5}	Z_{2Bit6}	Z_{2Bit7}
Byte 3	Z_{3Bit0}	Z_{3Bit1}	Z_{3Bit2}	Z_{3Bit3}	Z_{3Bit4}	Z_{3Bit5}	Z_{3Bit6}	Z_{3Bit7}
Byte 4	Z_{4Bit0}	Z_{4Bit1}	Z_{4Bit2}	Z_{4Bit3}	Z_{4Bit4}	Z_{4Bit5}	Z_{4Bit6}	Z_{4Bit7}
Byte 5	Z_{5Bit0}	Z_{5Bit1}	Z_{5Bit2}	Z_{5Bit3}	Z_{5Bit4}	Z_{5Bit5}	Z_{5Bit6}	Z_{5Bit7}
Byte 6	Z_{6Bit0}	Z_{6Bit1}	Z_{6Bit2}	Z_{6Bit3}	Z_{6Bit4}	Z_{6Bit5}	Z_{6Bit6}	Z_{6Bit7}
Byte 7	Z_{7Bit0}	Z_{7Bit1}	Z_{7Bit2}	Z_{7Bit3}	Z_{7Bit4}	Z_{7Bit5}	Z_{7Bit6}	Z_{7Bit7}
Byte 8	Z_{8Bit0}	Z_{8Bit1}	Z_{8Bit2}	Z_{8Bit3}	Z_{8Bit4}	Z_{8Bit5}	Z_{8Bit6}	Z_{8Bit7}

Thus, the total amount of errors over all the runs for this LET would be

$$Z_{total} = \sum_{i=1}^{8} \sum_{j=0}^{7} Z_{iBitj}$$

and thus the errors per second will be

$$E = Z_{total}/(\frac{M * 60}{R})$$

If an error would definitely occur, the probability of it occurring in any bit of any byte is

$$P_{iBitj} = \frac{Z_{iBitj}}{Z_{total}}$$

Now we can predict that E amount of errors will occur every second, with a probability of P_{iBitj} to be in a certain bit. From this, we can inject upsets by halting the core E amount of times, determine which bit the upset will be injected to, extract the byte value with JTAG, invert the bit where the upset takes place, and write the new byte to that SRAM cell. Then the core can resume, and the data injection is completed. Should E be greater than upsets can practically be injected per second, then E can be reduced. SEE testing with particle accelerators is an accelerated simulation of in-orbit SEE. Thus reducing the upset speed would be equivalent to reducing the beam current or the effective particle interactions per second.

Practically, systems are much larger than eight bytes. However, cumulative totals can be made of various groups to make the implementation more manageable. For instance, in the SRAM example, the probability of which byte the upset will occur in can be calculated first, and then a specific bit of the upset will be injected into.

This should give a more realistic injection pattern and could be used by designers to test their designs before SEE testing to see if it will be able to mitigate these effects. If their designs do not mitigate these effects, they can determine which areas need further mitigation. This can potentially help designers from not having to reperform SEE tests because of insufficient mitigation.

7.2 General Firmware Mitigation

With the addition of scripting, as described in Chapter 6, it is quite easy to test mitigation schemes against upsets, even without knowing the DUTs SEU sensitivity. With the

mitigation active, the mitigated areas can be targeted through automated injection. Faults can be induced while the DUT is running to see its functional effect, or an in-depth look at how the mitigation scheme works can be implemented by inducing a fault and stepping the core through JTAG several hundred or thousand steps forward and investigating if and how the fault propagates.

Through random injection of all active areas, the areas most sensitive to disruption if struck by an SEU can easily be determined. If these areas are critical, software mitigation can be applied, such as error-correcting codes, triple modular redundancy or majority voting. If the areas are not critical, lesser mitigation can be applied.

Fault injection can also be used to test built-in hardware error correction components. The chosen MCU for this project has a built-in CRC check. By enabling it and bombarding the area with induced faults, the resulting efficiency and additional overhead can be tested to determine if it is a valid mitigation tool. Further, benchmarks such as [63], which were specifically designed to detect and report SEUs, can be tested to determine their exact range of error detection and their effectiveness as benchmarks.

An implementation of this approach was tested where a basic SRAM fault detection program was loaded into the DUT firmware. The program scans a predetermined area of the SRAM that has been set to all zeros. If it finds any byte that is not all zeros, it increments a counter. After the entire area has been scanned, it reports the start address, the end address, and counter value over UART. If no faults are injected into the scanned SRAM cells, the program successfully reports no errors were found. When 100 errors are injected into the scanned SRAM cells, the program successfully reports that 100 errors have been found. So far, the program is working as expected.

But what happens if the area in memory where the counter is stored is also subjected to fault injection? Using TrueStudio, the memory location of the counter can be found. Faults were then injected into this location, and predictably the counter keeps reporting the wrong error values. As the UART output relies on the value stored inside the counter, this is expected, as that value is never verified to be correct before being transmitted. This means that this system has a vulnerability to upsets if the upset occurs in the same location the error counter is stored.

Using a technique called Triple Modular Redundancy (TMR), the DUT program was then modified to include three counters instead of just one. Every time an error is detected all counters are incremented. Before transmitting the error count, two of the counter values are compared with each other. If they are equal, then either an upset occurred in both memory locations that changed both values exactly the same way, which is very unlikely, or no upset

has occurred in either one of the two counters. However, if they are not equal, an upset has occurred in one of the two counters and the third can be used as a tiebreaker.

After this change, if one of the counter values experiences an upset, it will now be mitigated and not affect the output. This mitigation strategy's effectiveness can now also be tested by running the program and injecting errors into the counter memory locations. If the memory area the program scans is very small, it can finish scanning it quickly and there is less time for errors to occur. As a result, the TMR mitigation appears successful. If the area is very large, then scanning takes much longer, and TMR fails more constantly as multiple upsets affect multiple counter memory locations.

In this example, the counter memory address was specifically chosen as an injection point as it was obvious that it would be a vulnerability for data corruption, leading to a loss of some functionality of the program. However, with larger programs, it is much harder to pinpoint all of these vulnerable locations manually.

This is where the scripting, described in Chapter 6.4, becomes valuable. A predefined error injection probability distribution can be provided to the system with a higher chance of error injection where the tester feels important parts of the code is stored and a lower chance in unused or unimportant areas. A script can then be set up that repeatedly injects an error according to this distribution and waits a predetermined amount of time or parses the DUT UART output looking for errors. If the time between injections is enough to allow errors to completely propagate, then if the user sees an error, or the system discovers functionality loss from the UART output, the location that was last injected to is most likely vulnerable to upsets. If enough upsets are injected and propagated errors found, then a sensitivity map similar to the radiation sensitivity map can be determined, mapping the chance that a loss of functionality will occur if an upset is induced in that location.

Applying this approach to the TMR mitigated example above identified five sensitive areas in the SRAM, two with a large probability of loss in functionality, namely the starting address variable and the ending address variable, and three with a smaller probability of loss in functionality, namely the counter variables. Loss in functionality included small issues such as incorrect error detection counter or start or end address values over the UART output, to major system stalls where an injected error changed the value of the end address to smaller than that of the current address being scanned, and the loop exit condition could not be met.

Chapter 8
Conclusion

Overall, the goal of designing a test suit that allows testers to better prepare for SEE tests, verify their particular applied mitigation techniques before testing, help them determine cross-sections and SEU sensitivity for a range of MCUs, and boost their efficiency while performing SEE tests appears to have been mostly met.

This section briefly summarises the end product and recaps the process to get there. Then the research questions, asked in Chapter 1.5, are discussed. Finally, some recommendations for future work are given.

8.1 Summary of work

To reach the state the system is in now, research had to be done regarding SEE mechanics, SEE testing at specifically iTL, MCU options and technologies, fault injection as a test tool, and finally, how MCUs have been tested for SEE until now. This can all be found in Chapter 2.

Chapter 3 then merges and converts these research snippets into a practical concept design, and lays out the design drivers and major components of this project that would need to be implemented, such as the DUT communication and onboard testing, test suite adaptability and accessibility, the requirements for a testing station to control the DUT, and for a monitoring station that allows testers access to the system from safe distances.

This chapter then ends with the iterative design approach that was taken, which started in Chapter 4 with the design and implementation of the POC. The POC's goal was to determine if JTAG is viable as a data extraction and injection tool, as that will need to be a core component of the test suite. This was determined by wrapping the ST-Link CLI in a Python

script that controlled it to inject faults and extract data. It was found that data could be both written and read from the DUT, but the time taken to extract large quantities of data was simply too long, at 616s for the selected MCU. To limit this extraction time to only 10% of the total testing time would require exposure times of around 90 minutes.

Chapter 5 introduced the next iteration. Before any work is done on making the system SEE test ready, the ST-Link CLI driver was replaced by a custom, Python-based JTAG driver that could more effectively utilise the JTAG interface for this application's specific use case. This driver was reverse-engineered by monitoring the ST-Link CLI and the JTAG programmer's hardware interactions and lead to a significant increase in operational speed. This implementation was around 8.55 times faster than the POC, and for the same 10% overhead cap, SEE test exposures would only need to be a much more manageable 12 minutes. A dedicated testing station was then built that could connect the tester to the DUT via ethernet and communicate with the DUT over UART and JTAG. Additionally, a hardware interface was included that could control the power supply to the DUTs.

This iteration was then tested in a real SEE test at iTL. It physically integrated with the test setup as described by [15] with minimal effort, but there were some problems in the software implementation on the monitoring station that forced the testers to connect and operate the testing station manually.

In Chapter 6, these software implementation problems, namely the defective GUI and the incorrect SSH implementation in the Python script, were solved. The Python GUI was replaced by a Node.js interface that was easier to customise, more robust and also more portable, making it easy to package the entire application for easy use on different PCs. The incorrect SSH implementation was solved by resetting the Raspberry Pi used in the test station and swapping the GUI thread and the SSH thread so that the SSH communication would take main priority. A nifty feature that was also added was the ability to use scripts to execute commands automatically.

The final system is as described in Chapter 6. It consists of a monitoring station and a testing station that allows full control over compatible DUTs through UART communication and the JTAG interface. Communication between the monitoring station and test station is via SSH over ethernet, and a custom driver drives communication between the DUT and the test station. This driver was partially reverse-engineered from the original ST-Link CLI and allowed quick, effective automation of JTAG features.

All memory cells and peripheral registers that are accessible to the boundary scan register can be extracted to be analysed for SEUs. Depending on the DUT, most of these cells can be overwritten, allowing faults to be injected into the DUT to test the effect of SEU on the

device firmware. The MCU core can be halted, run, or stepped. The DUT can be power-cycled, and additional safety hardware can be added to protect against destructive effects like SEL. Most implemented modules on the test station can also be modified or completely replaced to broaden its use cases and to be adaptable.

Scripts can be used to automate the testing process, which is especially handy for fault injection. The final GUI appears to be working very smoothly and allows all aspects of the test suite to be controlled.

Chapter 7 then briefly explores using this final system as a dedicated fault injection tool. One approach is to determine which DUT areas are sensitive to SEU and then to mimic that pattern through fault injection for out-of-beam testing that simulates real SEE tests. This can allow testers to apply their mitigation techniques and test them before a real SEE test. Should there be a problem with the implementation, it can be fixed before the SEE test. Otherwise, another test would have been needed to retest the fixed version.

Another approach is to inject faults into the DUT to determine which areas are prone to functionality loss should an upset occur there. This allows for system weak spots to be identified and mitigated without any SEE sensitivity data. Areas can be identified by injecting faults and noting if a propagated fault is detected. Mitigation techniques can also be tested by focusing the injection to the mitigated area only, as in the example used.

8.2 Answering the research questions

This project aimed to answer several research questions. They will now be discussed and any answers found will be given

8.2.1 Which COTS MCUs should the local SEE testing be focused on?

The core of this question is investigated in Chapter 2.5.1. The conclusion was to focus on the STM32 Arm Cortex-M MCUs. These MCUs are similar and thus easily adaptable to equivalent Arm Cortex-M models from other manufactures. Chapter 2.6 and 2.7 show that previous research heavily tested Arm Cortex-M MCUs, and further a local satellite component design company showed specific interest in these MCUs.

Answer: The Arm Cortex-M MCU range, with this project focusing on the STM32L452RE

8.2.2 How can effective, low-level microprocessor SEE testing be integrated into the iTL environment?

The system developed in the work done from Chapter 4 to Chapter 6 allows testers to easily integrate their DUT to the SEE test methodology created in [15]. By implementing a system that allows fluent communication, especially for data extraction, to find low-level upset locations, testers can easily interface their DUTs inside the vault from outside the vault.

Further, SEE mitigation techniques can be tested before SEE testing through fault injection. This way, the tester knows what to expect from the DUT during the actual test, and can plan and test more effectively.

Answer: By implementing a system that conforms to the methodology of [15], allows both UART and JTAG data extraction and an easy interface between the DUT inside the vault and the tester outside, and that can aid testers in verifying that their planned tests are sensible before the SEE test itself.

8.2.3 How can tools like fault injection be used to help testers prepare for SEE testing and verify their mitigation techniques?

As seen in Chapter 2.6, fault injection is often used to test systems against the unknown. Chapter 7 discussed using fault injection in such a way as to emulate the SEE environment. The tester can subject the DUT to faults injected in a known, or guessed, distribution that can show how the beam could affect the DUT or where weak areas are that are sensitive to functional loss due to upsets. Mitigation techniques can also be tested by precisely injecting faults into the mitigated areas. These approaches all give the tester extra insight into how the DUT will react when irradiated and can help guide how SEE tests are conducted to be more effective.

Answer: Fault injection can be used to predict the system response of SEUs, which can aid testers to design effective tests and verify their mitigation techniques work properly before using valuable beam-time.

8.3 Improvements and recommendations

The current JTAG driver is very bare-bones. More complete drivers should replace it as they become available. The drivers developed by [67] and [68] are specifically for ST-Link V2-1 programmers as used in this project and should be considered.

The system should undergo another SEE test to find more weak spots. For future SEE tests, it might be worthwhile also running more involved firmware to push the data extraction speed limits in real-world use cases. It could also be viable to expand SEE test abilities to accommodate more aspects of TID testing, as TID also takes place during SEE testing, and as testers typically will schedule TID and SEE tests together due to logistical reasons.

The system might be improved by removing the ST-Link programmer and replacing it with a dedicated JTAG programmer. Dedicated JTAG programmers might be able to extract data quicker from the DUT, especially if they can make use of USB3.0. Also, during the SEE test the ST-Link programmer appeared to show more sensitivity to SEE than the DUT itself, though not enough data was collected to prove this distinctly. Using a dedicated JTAG programmer would allow larger distances between the programmer and the DUT, which should put the programmer further into the beam shadow. Dedicated JTAG programmers should also be able to expand the current implementation to all Arm Cortex-M4 processors and possibly the entire Arm Cortex-M range.

Finally, if a test approach as described in Chapter 7.1 is viable, it must be explored as it could significantly change the way MCU SEE testing could be done.

Appendices

Appendix A – Fluence Data for 29 January 2020

The following fluence report was generated by Dr A. Barnard, of Stellenbosch University, for the SEE test at iTL on 2020/01/29. This maps the measured fluence of the various runs throughout the test slot.

Report on BLM log data for SEE test at iTL on 2020/01/29

Prepared by
Arno Barnard

BLM log data plotted with observations and comments for each plot.
This data should be carefully interpreted while keeping the specific geometrical setup in mind.

A BLM active surface area of $7.34mm^2$ is used to convert fluence to per cm^2.

SEE Test Data generated at NRF iThemba LABS, Cape Town, South Africa.

Figure 1 shows logged values during BLM calibration. The logs indicated that the reference BLM was not generating pulses. A faulty BLM power cable was replaced.

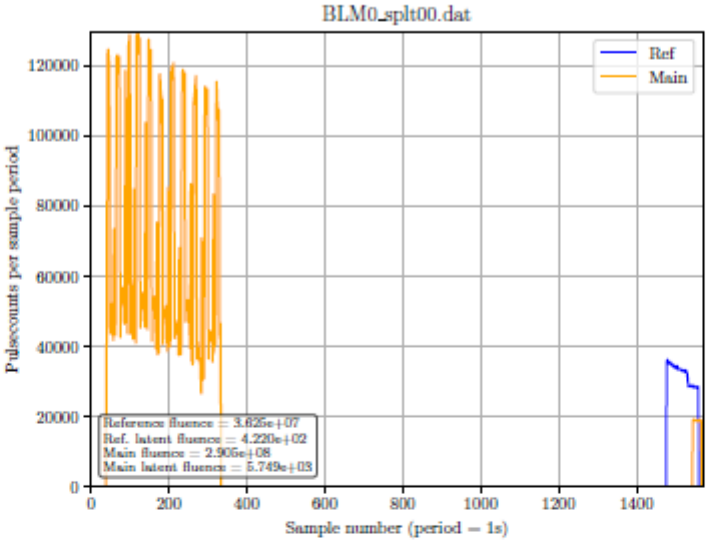


Figure 1: This is figure splt00.

Figure 2 shows the beam spot profile measurements with In-beam BLM in the beam spot. Measurements done at 2.4nA - 3.5nA (logbook is unclear)? Large periodic variations are visible.

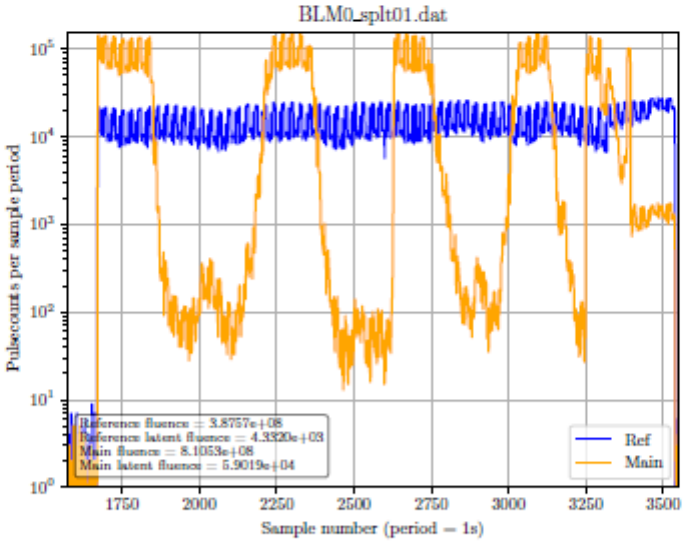


Figure 2: This is figure splt01.

Figure 3 shows 'Cup in' starting at 14:15. In-beam BLM still in spot.

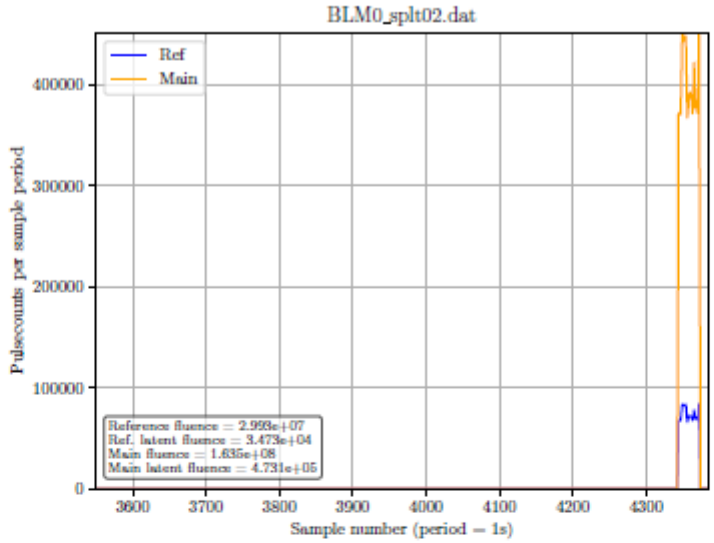


Figure 3: This is figure splt02.

Figure 4 shows 'Cup in' starting at 14:15. In-beam BLM still in spot.

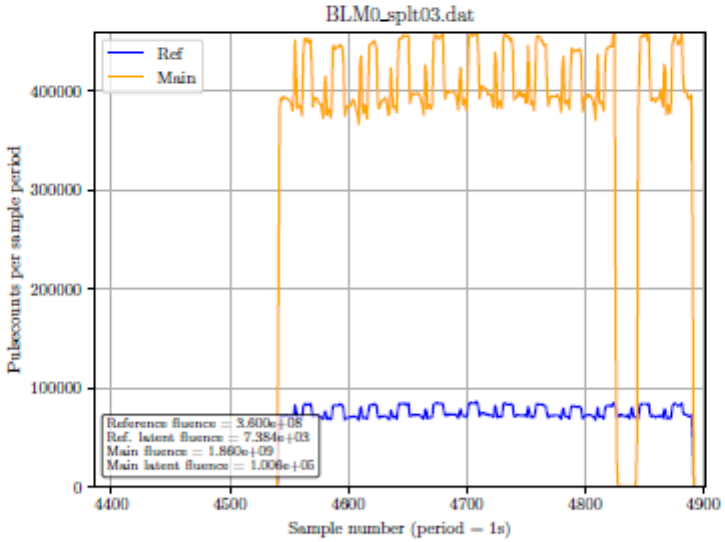


Figure 4: This is figure splt03.

At the start of Figure 5 (14:19), M. Tumwesigye's FPGA is moved into the beam spot until Figure 17, which stops at 17:16.

For the next three figures (Fig.5 - Fig.7) the current oscillation registered from 8.6 nA to 10.2 nA.

Figure 5 shows the ~1 min run starting at 14:19.

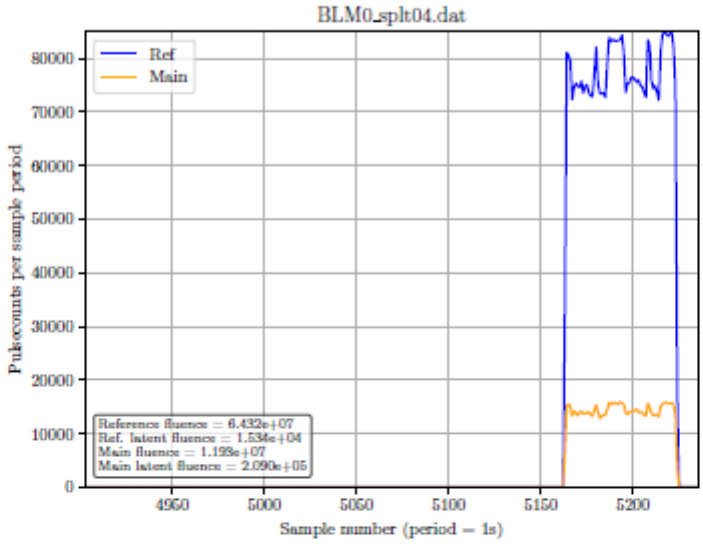


Figure 5: This is figure splt04.

Figure 6 shows the ~2 min run starting at 14:21.

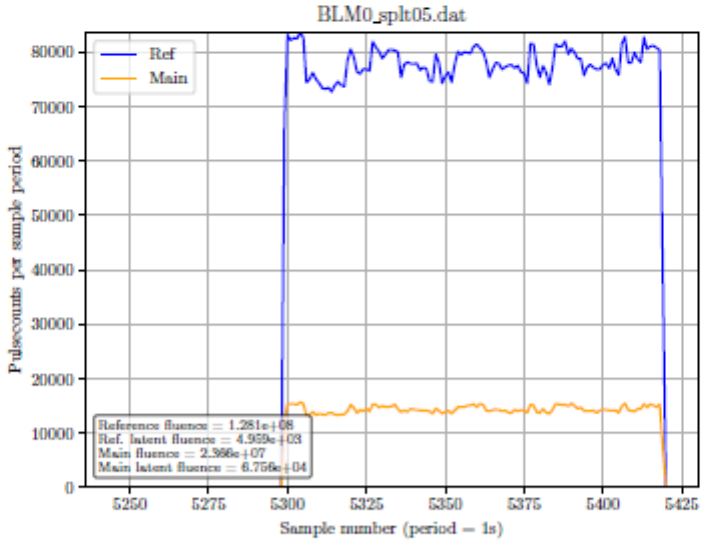


Figure 6: This is figure splt05.

Figure 7 shows the ~5 min run starting at 14:24.

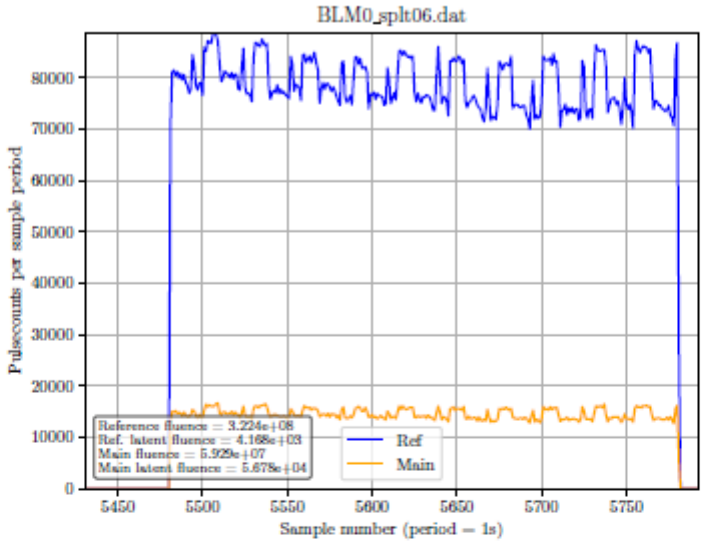


Figure 7: This is figure splt06.

For the next five figures (Fig.8 - Fig.12) the current oscillation registered from 18.2 nA to 21.3 nA. Figure 8 shows the ~2,15 min run starting at 14:32.

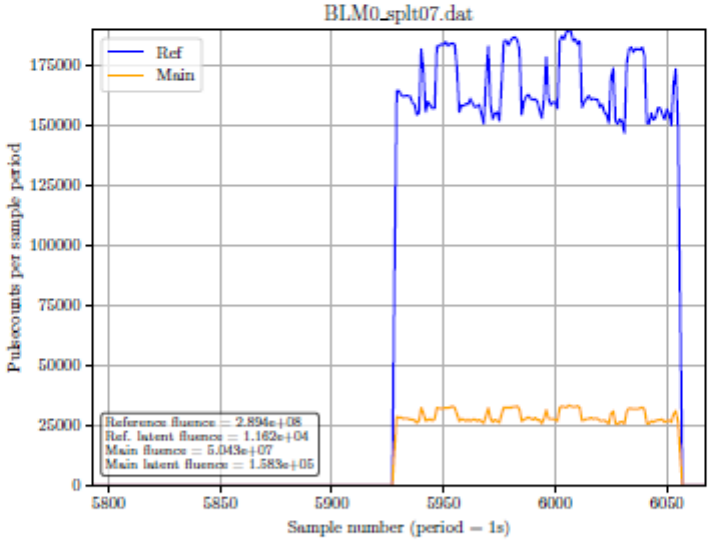


Figure 8: This is figure splt07.

Figure 9 shows the ∼2 min run starting at 14:35.

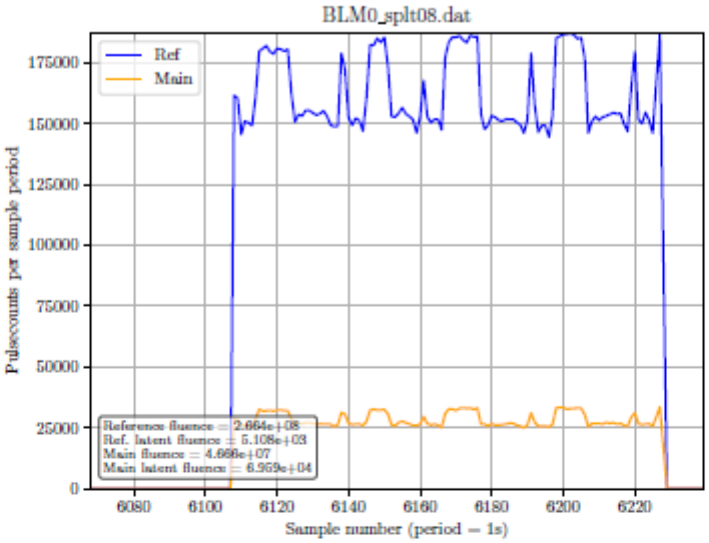


Figure 9: This is figure splt08.

Figure 10 shows the ∼5 min run starting at 14:38.

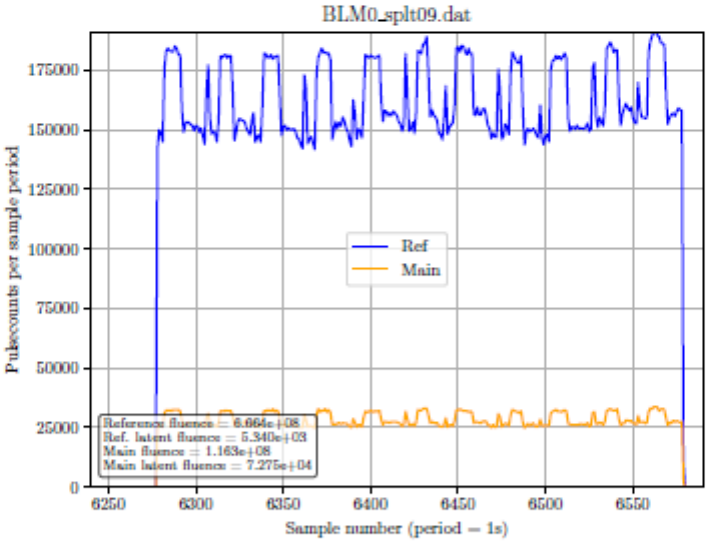


Figure 10: This is figure splt09.

Figure 11 shows the ~5 min run starting at 14:45.

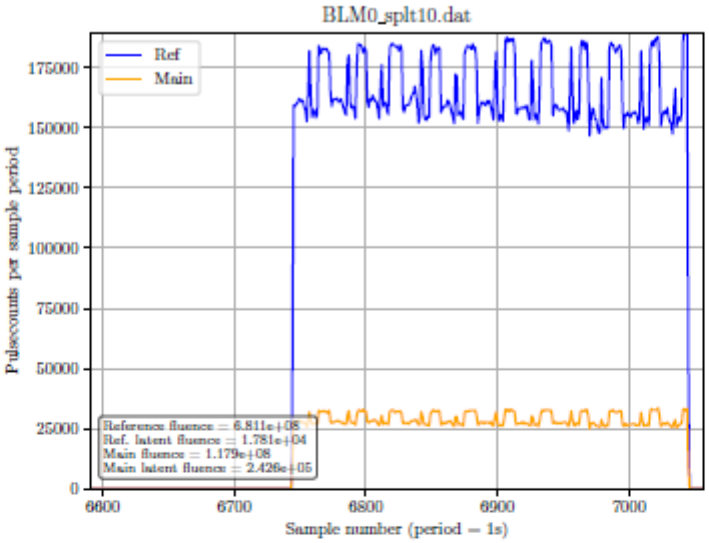


Figure 11: This is figure splt10.

Figure 12 shows the ~5 min run starting at 14:51.

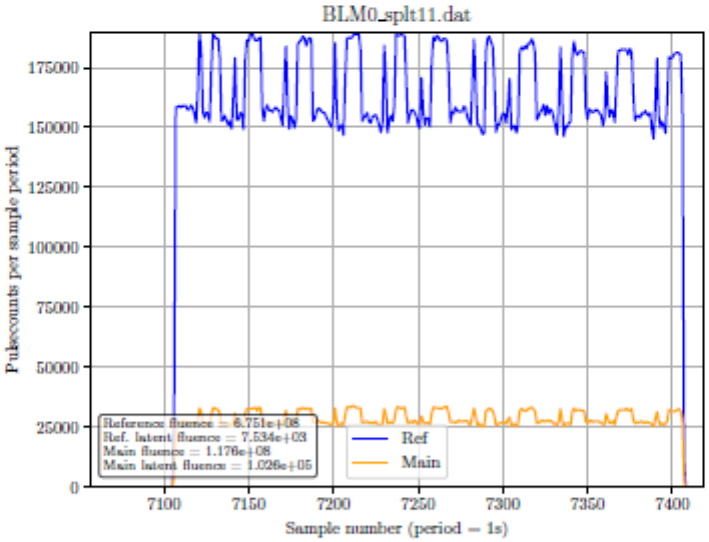


Figure 12: This is figure splt11.

Beam current is changed to ∼40 nA for the next two runs (Fig.13, Fig.14).
Figure 13 shows the ∼30 s run starting at 14:59.

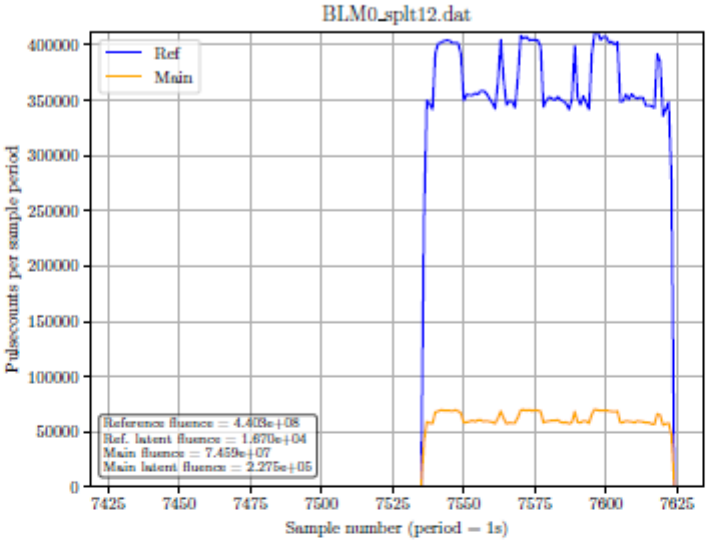


Figure 13: This is figure splt12.

Figure 14 shows the ∼1 min 40 s run starting at 15:00.

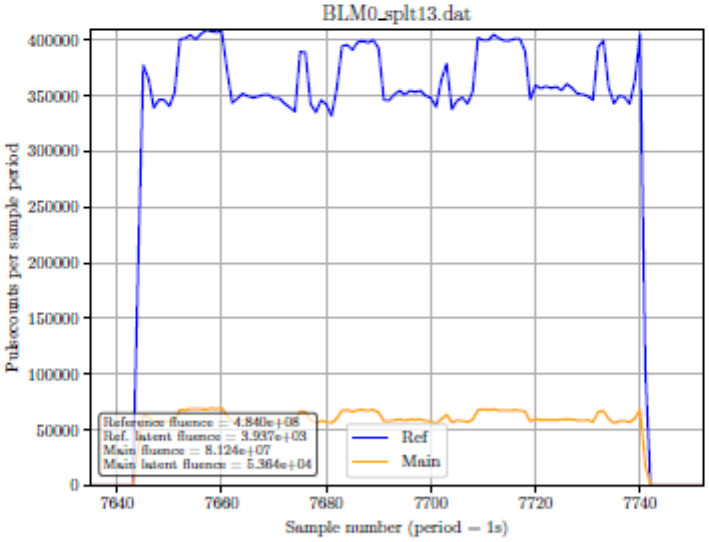


Figure 14: This is figure splt13.

Bean current is set back to 8.6 nA - 10.2 nA.
Figure 15 shows the ∼10 min run starting at 15:09.

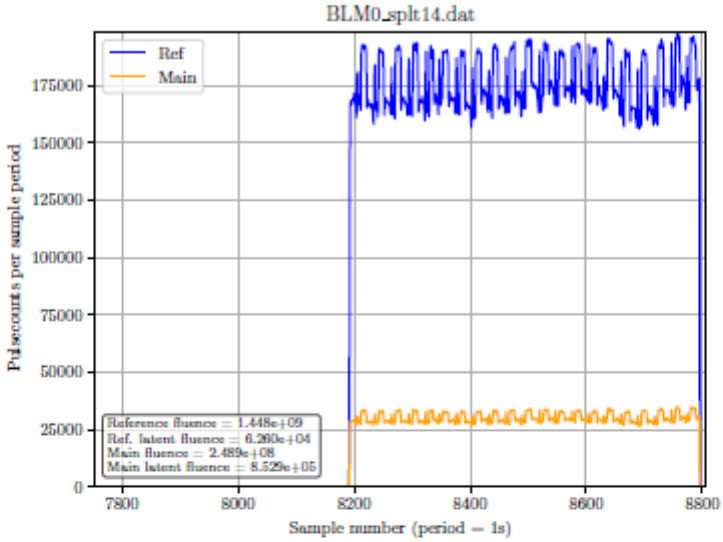


Figure 15: This is figure splt14.

Figure 16 shows the ∼90 min run (9 x 10 min entries in logbook) starting at 15:23 and ending at 16:53.

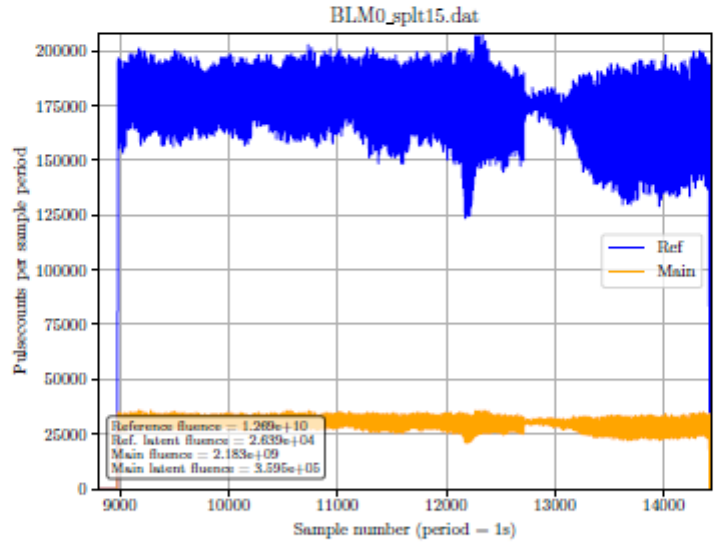


Figure 16: This is figure splt15.

Beam current is set to 40 nA.
Figure 17 shows the ∼10 min run starting at 17:06.

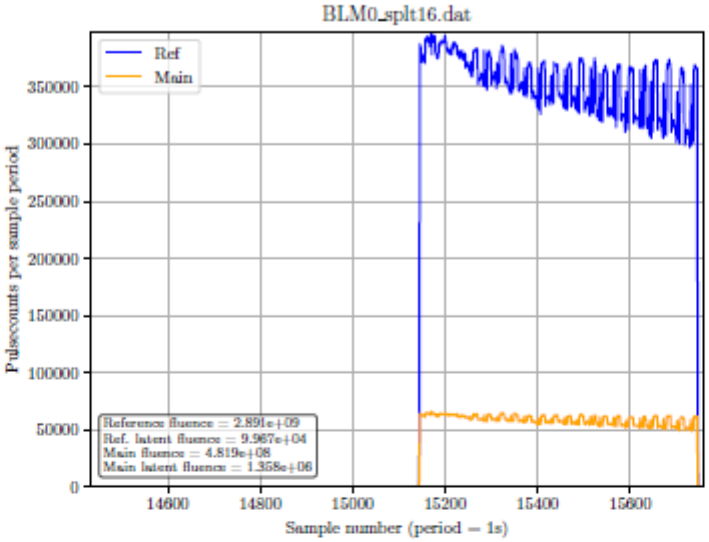


Figure 17: This is figure splt16.

V. Sciocatti's processor is moved into the beam spot until 19:00.
Beam current is set to 20 nA for next three runs.
Figure 18 shows the ∼2 min run starting at 17:48.

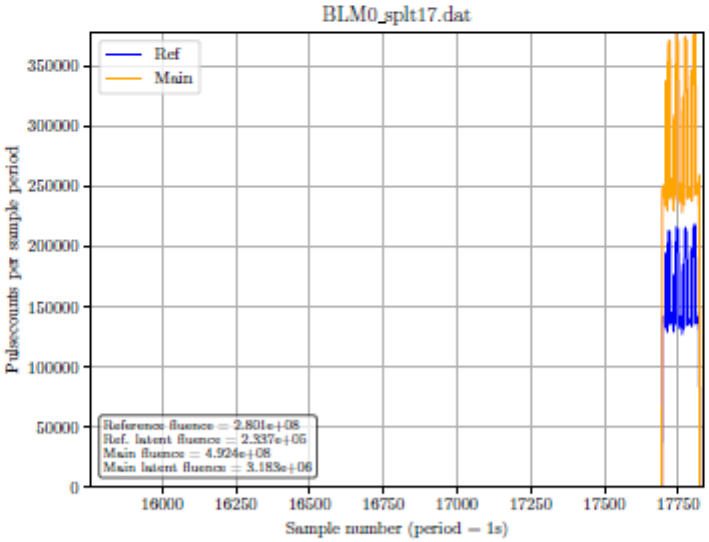


Figure 18: This is figure splt17.

Figure 19 shows the ~5 min run starting at 17:53.

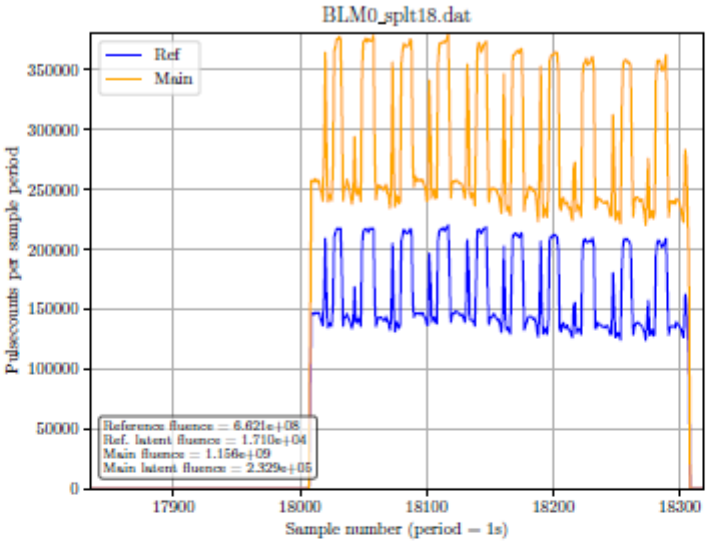


Figure 19: This is figure splt18.

Figure 20 shows the ~10 min run starting at 18:02.

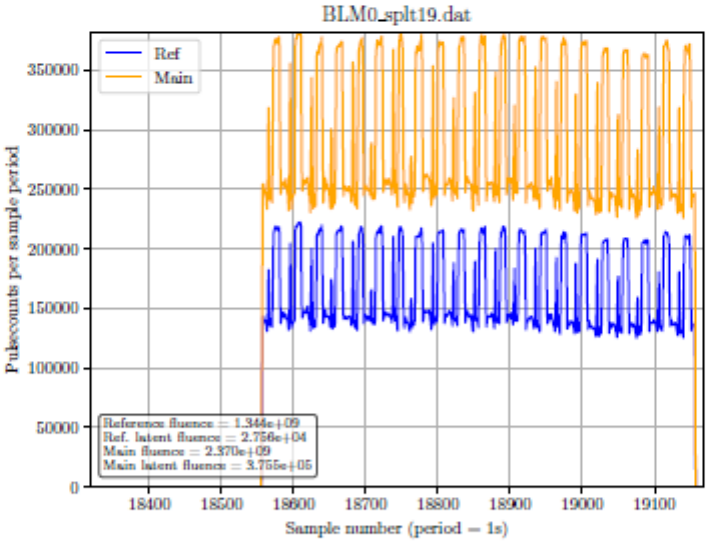


Figure 20: This is figure splt19.

Beam current is set to 41 nA for the next four runs.
Figure 21 shows the ∼10 min run starting at 18:19.

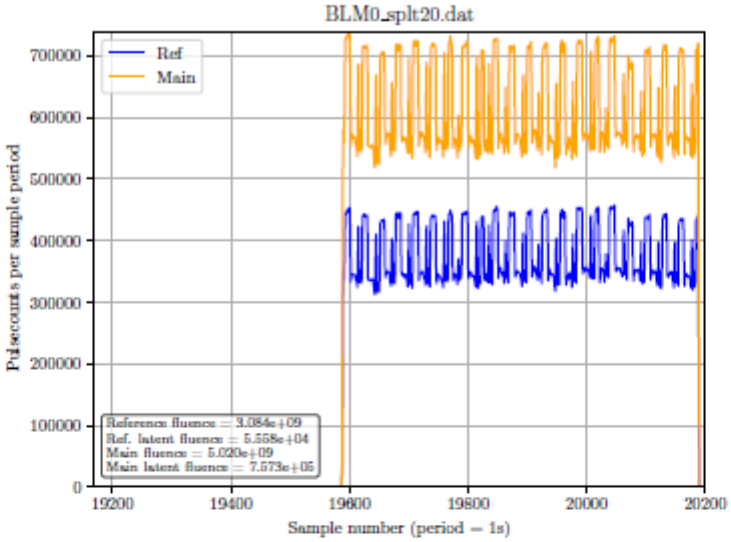


Figure 21: This is figure splt20.

Figure 22 shows the ∼10 min run starting at 18:37.

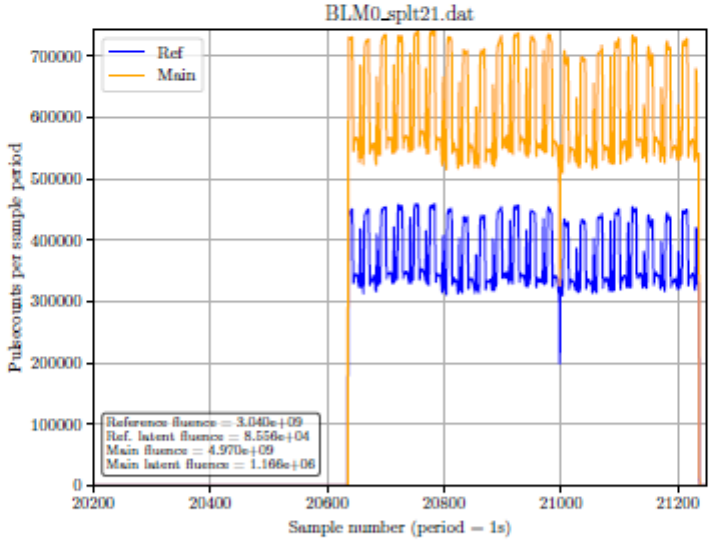


Figure 22: This is figure splt21.

Figure 23 shows the ~10 min run starting at 18:50.

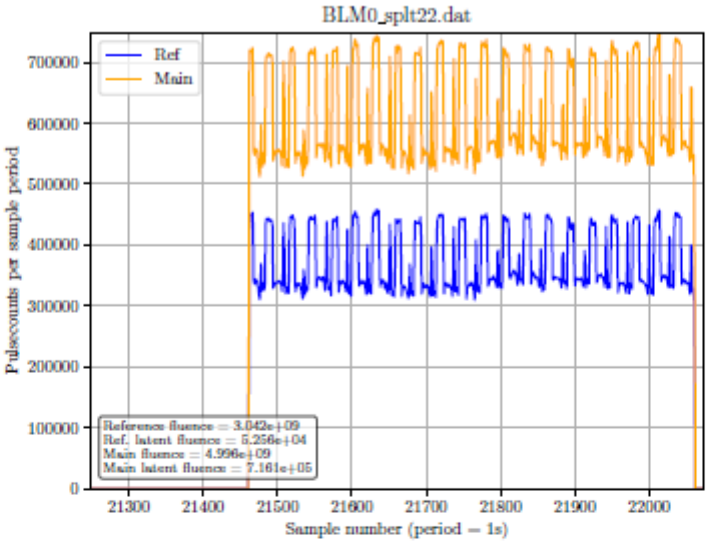


Figure 23: This is figure splt22.

M. Tumwesigye's FPGA is moved into the beam spot.
Figure 24 shows the ~30 min run starting at 19:04.

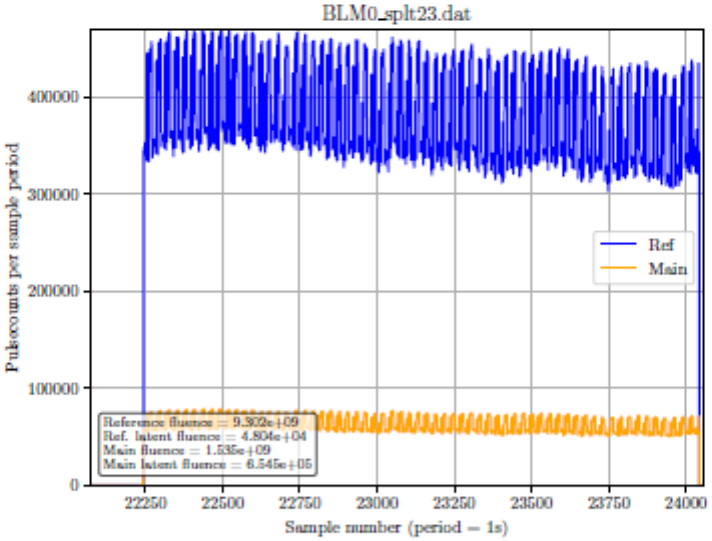


Figure 24: This is figure splt23.

Figure 25 shows the second beam profile measurement done at 41 nA.

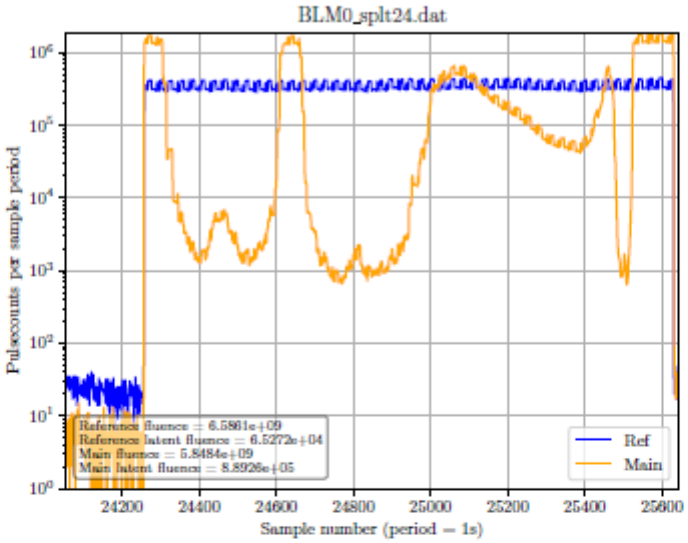


Figure 25: This is figure splt24.

Figure 26 shows the current calibration (15 nA - 25 nA).

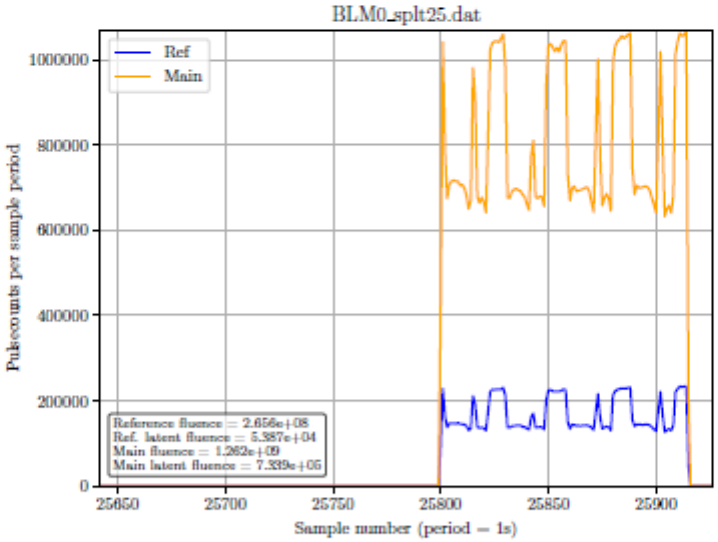
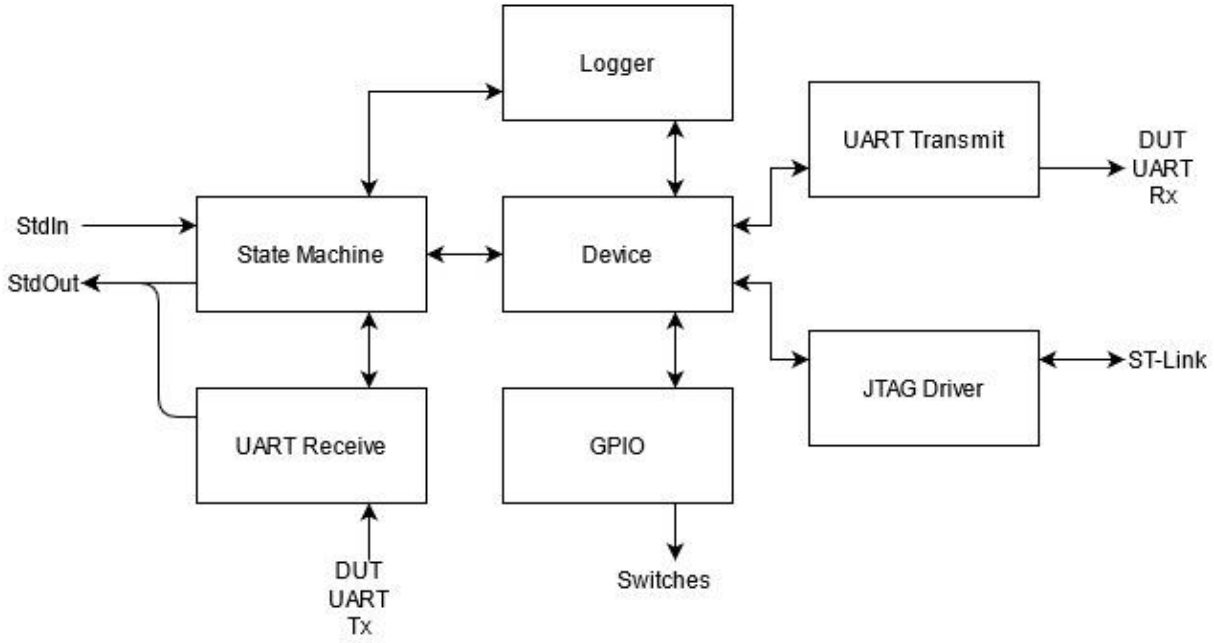


Figure 26: This is figure splt25.

Appendix B – Final Implementation APIs

This Appendix lists the final API implementation as it stands at the end of this project. The modules are shown in the figure below, and each module has specific functions that can be modified. Of course, even these functions can be removed and changed, but then it is up to the programmer to ensure all modules are still working together as needed.



State Machine

The state machine module runs in its own thread and manages the control flow of the program. It can parse instructions from StdIn, and then delegate to the other modules. Instructions are handled in a FIFO manner.

Function Name	Parameters	Output	Description
report(msg)	msg – the message to report	None	Sends a message over StdOut, user gets notified. Only use when important.
status(cls, stat)	cls – the class instance that sends the status stat – the status	None	Sends a status over StdOut, used by receiver to proceed.
execute(json)	Json – the command data to execute	None	Executes the json command as it

			would execute a command received over StdIn. This allows other local modules to interact.

UART Receive

The UART Receive module can listen for incoming messages from an active virtual COM port. It can also filter the input using a user-defined filter

Function Name	Parameters	Output	Description
listen(device)	device – the device to listen for.	bool – status of operation success	If any messages are received from the device, json-format and print them to StdOut so they go directly to the user.
deafen(device)	device – the device to stop listening for.	bool – status of operation success	Ignore messages received from device.
filterON(filter, device)	filter – the name of the filter to implement. device – the device on which input to apply the filter to.	bool – status of operation success	Apply filter to messages from device. Can be coupled with "execute(json)" of the state machine to influence the rest of system.
filterOFF(device)	device – the device on which to stop applying the filter to.	bool – status of operation success	Stop applying a filter on the device input.

Logger

The logger module organises and creates log files, and also manages the writing to them. This prevents multiple modules trying to write to the same file at once. This module also requests SFTP transfers when important data needs to be moved from the test station to the monitoring station.

Function Name	Parameters	Output	Description
create_log()	None	bool – status of operation success	Creates a directory for logging in the local folder, on two flash drives, and on the user PC. All runs will be placed in these folders until this function is run again. Also creates the base log files.
create_run()	None	bool – status of operation success	Creates a run file. Run files are used to group data extracted from the DUT in an easily parsable manner.
log(data)	data – extracted data to add to a run log	bool – status of operation success	Add data to a run file.
log(detail)	detail – operational details to log	bool – status of operation success	Log any detail that cannot be inside a run file, such as status messages or user notes.

UART Transmit

This module detects available virtual COM ports, connects to them, and transmits any messages to them via UART. Every connection has its own instance.

Function Name	Parameters	Output	Description
listUART()	None	a list of all available COM ports	Detects all available virtual COM ports
connectUART(number, json)	number – the index of the device in the list() output list to connect to. json – JSON data containing the	bool – status of operation success	Connects to the specified virtual COM port

	needed connection parameters.		
transmit(msg)	msg – the message to transmit	bool – status of operation success	Transmits a message over the virtual COM port.

JTAG Driver

This module links the rest of the application to the DUT. Each DUT has its own instance, initialised with the connection parameters(if they are provided).

Function Name	Parameters	Output	Description
listJTAG()	None	list of available JTAG devices.	Finds and returns a list of available JTAG devices.
connectJTAG(params)	params – the parameters, in json, to connect to.	bool – status of operation success	Connects to a specific JTAG device.
read(start, size)	start – start address to read. size – size of area to read.	array of extracted data.	Read a data chunk from a device, starting at start, with a width of size.
readAll()	None	bool – status of operation success	Extracts all data defined in a predetermined map to a file.
write(address, data)	address – address to write data to. data – value to write to data register.	bool – status of operation success	Writes data to the device at address.
writePattern(pattern)	pattern – selected pattern from predefined list.	bool – status of operation success	Writes a predetermined pattern into the device.
coreHalt()	None	bool – status of operation success	Halts the core.
coreResume()	None	bool – status of operation success	Resumes the core.
coreStep()	None	bool – status of operation success	Steps the core (must be halted first).

getRegisters()	None	array of register values.	Reads all the CPU registers. Core must be halted first.
setRegister(register, data))	register – the register to write the data to. data – the value to write into the register.	bool – status of operation success	Set a specific register value. Core must be halted first.

GPIO

This module controls the GPIO pins. Currently it is only configured to switch the power to DUTs on or off. Instances are initialised with a pin number.

Function Name	Parameters	Output	Description
startGPIO()	None	bool – status of operation success	Initialize the GPIO for use. Called when program starts.
stopGPIO()	None	bool – status of operation success	Reset the GPIO so that it is in default configuration for the next program to use.
connect(pin)	pin – the pin the instance is linked to.	bool – status of operation success	Changes the connect pin to the parameter pin.
pinOn()	None	bool – status of operation success	Switches pin state to on.
pinOff()	None	bool – status of operation success	Switches pin state to off.

Device

The Device module links all the hardware modules (JTAG driver, UART transmit, GPIO) into a single instance. All functions can be called of the sub-modules simply by using their original names, ie. device.pinOff() will switch off its connected GPIO pin. This just makes it conceptually easier to manage. It does imply that all functions in the sub-modules need to be uniquely named.

References

[1] W. H. Steyn, R. van Zyl, M. Inggs and P. J. Cilliers, "Current and future small satellite projects in South Africa," *2013 IEEE International Geoscience and Remote Sensing Symposium - IGARSS*, Melbourne, VIC, Australia, 2013, pp. 1294-1297, doi: 10.1109/IGARSS.2013.6723018.

[2] "ZACube-1 – South African Council for Space Affairs", *Sacsa.gov.za*, 2019. [Online]. Available: http://www.sacsa.gov.za/zacube-1/. [Accessed: 4- Mar- 2019].

[3] R. Nevhulaudzi, "South African space sector set to grow with new Space Infrastructure Hub | SANSA", *SANSA*, 2021. [Online]. Available: https://www.sansa.org.za/2020/08/27/south-african-space-sector-set-to-grow-with-new-space-infrastructure-hub/. [Accessed: 16- Feb- 2021].

[4] "Space in South Africa | Space Lab", *Spacelab.uct.ac.za*, 2020. [Online]. Available: http://www.spacelab.uct.ac.za/space-south-africa-0. [Accessed: 25- Apr- 2020].

[5] V. Strakos, "Technological and design limitations of bipolar power transistors," *2009 32nd International Spring Seminar on Electronics Technology*, Brno, Czech Republic, 2009, pp. 1-3, doi: 10.1109/ISSE.2009.5206928.

[6] J. R. Azambuja, Â. Lapolli, L. Rosa and F. L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique," *in IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 993-1000, June 2011, doi: 10.1109/TNS.2011.2109398.

[7] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, pp. 6–7, 2005

[8] R. C. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources," in IEEE Transactions on Device and Materials Reliability, vol. 1, no. 1, pp. 17-22, March 2001, doi: 10.1109/7298.946456.

[9] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I.C. J. Montrose, H. W. Curtis, and J. L. Walsh, "Field testing for cosmic ray soft errors in semiconductor memories," *IBM J. Res. Develop.*, pp.41–49, Jan. 1996

[10] "SEECA - Section 1", *Radhome.gsfc.nasa.gov*, 2019. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/papers/seeca1.htm. [Accessed: 21- May- 2019].

[11] P. E. Dodd, "Physics-based simulation of single-event effects," in *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 343-357, Sept. 2005, doi: 10.1109/TDMR.2005.855826.

[12] "SEECA - Section 4", *Radhome.gsfc.nasa.gov*, 2019. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/papers/seeca4.htm. [Accessed: 21- May- 2019].

[13] R. Edwards, "Technical specification for atmospheric radiation single event effects (SEE) on avionics electronics," *The IEE Seminar on Cosmic Radiation Single Event Effects and Avionics, 2005 (Ref. No. 2005/11270)*, London, UK, 2005, pp. 6-6/21, doi: 10.1049/ic:20050512.

[14] Petersen, E.: Single event effects in aerospace. Wiley-IEEE Press, 2011. ISBN 9780470767498. 1.2

[15] Barnard, Arno. "Establishing a Viable and Reliable Proton-Induced Single Event Effect Test Methodology and Environment at iThemba LABS in South Africa." Stellenbosch : Stellenbosch University, 2020. Print.

[16] S. Buchner, D. McMorrow, "Overview of Single Event Effects", *presented at SERESSA* 2015, Puebla, Mexico, 2015

[17] Development of a Digital Photometer for the Evaluation of COTS LED Fast Neutron Dosimeter; Integrated Measurement Systems for Electronic Devices Operating in Radiation Environment - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Example-of-Single-Event-Effect-Single-Event-Upset-20_fig2_320126769 [accessed 5- May- 2019]

[18] "Cross section (physics)", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Cross_section_(physics). [Accessed: 23- May- 2019].

[19] "Speak physics: What is a cross section?", *symmetry magazine*, 2019. [Online]. Available: https://www.symmetrymagazine.org/article/speak-physics-what-is-a-cross-section. [Accessed: 23- May- 2019].

[20] D. MacManus, "Linear energy transfer | Radiology Reference Article | Radiopaedia.org", *Radiopaedia.org*, 2019. [Online]. Available: https://radiopaedia.org/articles/linear-energy-transfer. [Accessed: 23- May- 2019].

[21] R. H. Maurer, M. E. Fraeman, M. N. Martin, D. R. Roth, "Harsh Environments: Space Radiation Environment, Effects, and Mitigation," *Johns Hopkins APL Technical Digest*, Volume 28, Number 1 (2008)

[22] MIL-STD-883E METHOD 1019.4 IONIZING RADIATION (TOTAL DOSE) TEST PROCEDURE, 15 November 1991

[23] K. Ma *et al.*, "Radiation-Induced Degradation Analysis and Reliability Modeling of COTS ADCs for Space-Borne Miniature Fiber-Optic Gyroscopes," in *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1-8, 2021, Art no. 9505908, doi: 10.1109/TIM.2021.3054419.

[24] A. Barnard and W. H. Steyn, "Low cost TID testing of COTS components," *2007 9th European Conference on Radiation and Its Effects on Components and Systems*, Deauville, France, 2007, pp. 1-4, doi: 10.1109/RADECS.2007.5205562.

[25] H. -. Wang, R. Liu, X. -. Li, L. Chen, D. M. Hiemstra and V. Kirischian, "Total ionizing dose test facilities for micro-electronic circuits," *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Vancouver, BC, Canada, 2016, pp. 1-4, doi: 10.1109/CCECE.2016.7726602.

[26] "Accelerators | CERN", *Home.cern*, 2019. [Online]. Available: https://home.cern/science/accelerators. [Accessed: 17- Jun- 2019].

[27] "Guide for Heavy Ion Radiotherapy", *Particle.or.jp*, 2019. [Online]. Available: https://www.particle.or.jp/hirtjapan/en/what/. [Accessed: 16- Sep- 2019].

[28] P. E. Dodd *et al.*, "Impact of Heavy Ion Energy and Nuclear Interactions on Single-Event Upset and Latchup in Integrated Circuits," in *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2303-2311, Dec. 2007, doi: 10.1109/TNS.2007.909844.

[29] I. V. Kalagin *et al.*, "Using the FLNR Accelerator Complex for SEE Testing: State of Art and Future Development," *2015 15th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, Moscow, Russia, 2015, pp. 1-6, doi: 10.1109/RADECS.2015.7365692.

[30] Weber, U. and Kraft, G., 2021. *Comparison of Carbon Ions Versus Protons.* [online] Available at: https://pubmed.ncbi.nlm.nih.gov/19672150/. [Accessed 4 Nov 2019].

[31] "PTCOG - Facilities in Operation", *Ptcog.ch*, 2019. [Online]. Available: https://www.ptcog.ch/index.php/facilities-in-operation. [Accessed: 4- Nov- 2019].

[32] Weber U, Kraft G. "Comparison of carbon ions versus protons." *Cancer J. 2009 Jul-Aug;*15(4):325-32. doi: 10.1097/PPO.0b013e3181b01935. PMID: 19672150.

[33] J. R. Schwank *et al.*, "Hardness Assurance Testing for Proton Direct Ionization Effects," in *IEEE Transactions on Nuclear Science*, vol. 59, no. 4, pp. 1197-1202, Aug. 2012, doi: 10.1109/TNS.2011.2177862.

[34] A. V. Prokofiev *et al.*, "A New Neutron Beam Facility for SEE Testing," *2005 8th European Conference on Radiation and Its Effects on Components and Systems*, Cap d'Agde, France, 2005, pp. PW14-1-PW14-4, doi: 10.1109/RADECS.2005.4365654.

[35] ECSS: Single Event Effects Test Method and Guidelines - ESCC Basic Specification No . 25100 Issue 2.ESA Requirements and Standards Division, 2014. 1.

[36] Lauenstein, J.: Standards for Radiation Effects Testing: Ensuring Scientific Rigor in the Face ofBudget Realities and Modern Device Challenges. 2015.Available at:http://ntrs.nasa.gov/search.jsp?R=201500114622.6, 2.3, 2.4

[37] "SAINTS@tlabs Partner Institutions (international) | iThemba LABS", *iThemba LABS*, 2021. [Online]. Available: https://tlabs.ac.za/saints/partner-institutions-international/. [Accessed: 22- Jan- 2020].

[38] "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices", *Analog.com*, 2020. [Online]. Available:

https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#. [Accessed: 28- Jan- 2020].

[39] S. Corrigan, "Introduction to the Controller Area Network (CAN)", TI.com, 2016. [Online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1612678352341&ref_url=https%253A%252F%252Fwww.google.com%252F. [Accessed: 4- Mar- 2019]

[40] "ISO11898-2.svg", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/File:ISO11898-2.svg. [Accessed: 29- Jan- 2020].

[41] A. Scholz, T. Hsiao, J. Juang, C. Cherciu, "Open source implementation of ECSS CAN bus protocol for CubeSats", *Advances in Space Research*, Volume 62, Issue 12, 2018, Pages 3438-3448, ISSN 0273-1177, https://doi.org/10.1016/j.asr.2017.10.015.

[42] "Introduction to SPI Interface | Analog Devices", *Analog.com*, 2020. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html. [Accessed: 29- Jan- 2020].

[43] "CAN bus", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus. [Accessed: 29- Jan- 2020].

[44] S. Campbell, "Basics of the I2C Communication Protocol", *Circuit Basics*, 2020. [Online]. Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/. [Accessed: 29- Jan- 2020].

[45] "Serial Peripheral Interface (SPI) - learn.sparkfun.com", *Learn.sparkfun.com*, 2020. [Online]. Available: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all. [Accessed: 29- Jan- 2020].

[46] "Analog-to-Digital Converters (ADCs) | Overview | Data Converters | TI.com", *Ti.com*, 2020. [Online]. Available: https://www.ti.com/data-converters/adc-circuit/overview.html. [Accessed: 29- Jan- 2020].

[47] D. Jr, "A question about 16 bit ADC representation", *Electrical Engineering Stack Exchange*, 2020. [Online]. Available: https://electronics.stackexchange.com/questions/387466/a-question-about-16-bit-adc-representation. [Accessed: 29- Jan- 2020].

[48] Arm Ltd., "Microprocessor Cores and Technology – Arm", *Arm | The Architecture for the Digital World*, 2020. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu. [Accessed: 31- Mar- 2020].

[49] "Securing the JTAG Interface | ASSET InterTech", *ASSET InterTech*, 2020. [Online]. Available: https://www.asset-intertech.com/resources/blog/2019/07/securing-the-jtag-interface. [Accessed: 31- Mar- 2020].

[50] "What is JTAG and how can I make use of it? - XJTAG Tutorial", *XJTAG*, 2020. [Online]. Available: https://www.xjtag.com/about-jtag/what-is-jtag/. [Accessed: 31- Mar- 2020].

[51] "JTAG", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/JTAG#Debugging. [Accessed: 31- Mar- 2020].

[52] "Boundary-Scan – JTAG", *JTAG*, 2020. [Online]. Available: https://www.jtag.com/boundary-scan/. [Accessed: 31- Mar- 2020].

[53] "Cisco 12000 Single Event Upset Failures Overview and Work Around Summary", *Cisco*, 2021. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/field-notices/200/fn25994.html. [Accessed: 16- Jan- 2021].

[54] D. M. Hiemstra and A. Baril, "Single event upset characterization of the Pentium(R) MMX and Pentium(R) II microprocessors using proton irradiation," in *IEEE Transactions on Nuclear Science*, vol. 46, no. 6, pp. 1453-1460, Dec. 1999, doi: 10.1109/23.819107.

[55] D. M. Hiemstra and A. Baril, "Single event upset characterization of the Pentium(R) MMX and Celeron(R) microprocessors using proton irradiation," *2000 IEEE Radiation Effects Data Workshop. Workshop Record. Held in conjunction with IEEE Nuclear and Space Radiation Effects Conference (Cat. No.00TH8527)*, Reno, NV, USA, 2000, pp. 39-44, doi: 10.1109/REDW.2000.896267.

[56] D. M. Hienistra, S. Yu and M. Pop, "Single event upset characterization of the Pentium(R) MMX and low power Pentium(R) MMX microprocessors using proton irradiation," *2001 IEEE Radiation Effects Data Workshop. NSREC 2001. Workshop Record. Held in conjunction with IEEE Nuclear and Space Radiation Effects Conference (Cat. No.01TH8588)*, Vancouver, BC, Canada, 2001, pp. 32-37, doi: 10.1109/REDW.2001.960445.

[57] G. M. Swift, F. F. Fannanesh, S. M. Guertin, F. Irom and D. G. Millward, "Single-event upset in the PowerPC750 microprocessor," in *IEEE Transactions on Nuclear Science*, vol. 48, no. 6, pp. 1822-1827, Dec. 2001, doi: 10.1109/23.983136.

[58] M. Cabanas-Holmen, E. H. Cannon, A. Kleinosowski, J. Ballast, J. Killens and J. Socha, "Clock and Reset Transients in a 90 nm RHBD Single-Core Tilera Processor," in *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3505-3510, Dec. 2009, doi: 10.1109/TNS.2009.2034314.

[59] H. Quinn, T. Fairbanks, J. L. Tripp, G. Duran and B. Lopez, "Single-Event Effects in Low-Cost, Low-Power Microprocessors," *2014 IEEE Radiation Effects Data Workshop (REDW)*, Paris, France, 2014, pp. 1-9, doi: 10.1109/REDW.2014.7004596.

[60] T. Fairbanks, H. Quinn, J. Tripp, J. Michel, A. Warniment and N. Dallmann, "Compendium of TID, Neutron, Proton and Heavy Ion Testing of Satellite Electronics for Los Alamos National Laboratory," *2013 IEEE Radiation Effects Data Workshop (REDW)*, San Francisco, CA, USA, 2013, pp. 1-6, doi: 10.1109/REDW.2013.6658191.

[61] H. Quinn, T. Fairbanks, J. L. Tripp and A. Manuzzato, "The Reliability of Software Algorithms and Software-Based Mitigation Techniques in Digital Signal Processors," *2013 IEEE Radiation Effects Data Workshop (REDW)*, San Francisco, CA, USA, 2013, pp. 1-8, doi: 10.1109/REDW.2013.6658218.

[62] F. Irom, "Guideline for ground radiation testing of microprocessors in the space radiation environment," *Jet Propulsion Laboratory, Tech. Rep.* 13, 2008.

[63] "lanl/benchmark_codes", *GitHub*, 2020. [Online]. Available: https://github.com/lanl/benchmark_codes. [Accessed: 16- Oct- 2020].

[64] J. R. Azambuja, Â. Lapolli, L. Rosa and F. L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique," in *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 993-1000, June 2011, doi: 10.1109/TNS.2011.2109398.

[65] "ST-LINK/V2 - STMicroelectronics", *STMicroelectronics*, 2020. [Online]. Available: https://www.st.com/en/development-tools/st-link-v2.html. [Accessed: 31- Mar- 2020].

[66] "libusb", *Libusb.info*, 2020. [Online]. Available: https://libusb.info/. [Accessed: 16- Feb- 2020].

[67] "disk91/PySWD", *GitHub*, 2020. [Online]. Available: https://github.com/disk91/PySWD. [Accessed: 16- Feb- 2020].

[68] "cortexm/pyswd", *GitHub*, 2020. [Online]. Available: https://github.com/cortexm/pyswd/tree/v2. [Accessed: 16- Feb- 2020].

[69] "Flash Key Register (Flash_Keyr); Flash Option Key Register (Flash_Optkeyr) - ST STM32F205 Programming Manual [Page 21] | ManualsLib", *Manualslib.com*, 2020. [Online]. Available: https://www.manualslib.com/manual/1586275/St-Stm32f205.html?page=21. [Accessed: 16- Feb- 2020].

[70] "Welcome to Paramiko! — Paramiko documentation", *Paramiko.org*, 2019. [Online]. Available: http://www.paramiko.org/. [Accessed: 16- Apr- 2019].

[71] *Cubesatshop.com*, 2021. [Online]. Available: https://www.cubesatshop.com/wp-content/uploads/2016/06/CubeComputer-V4.1-Datasheet-v1.3.pdf. [Accessed: 26- Feb- 2020].

[72] "USB 3.0", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/USB_3.0#USB_3.2. [Accessed: 26- Feb- 2021].

[73] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson and E. Encrenaz, "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller," *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Los Alamitos, CA, USA, 2013, pp. 77-88, doi: 10.1109/FDTC.2013.9.

[74] Arm Ltd., "CoreSight Architecture | Serial Wire Debug – Arm Developer", *Arm Developer*, 2021. [Online]. Available: https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug. [Accessed: 26- Feb- 2021].

[75] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proceedings of the IEEE, 100(11):3056–3076, 2012.

[76] J. Juterte, "Hardware attacks: theory and experimental state-of-the-art of laser fault injection attacks", *Journees Nationales 2018*, 2018. [Online]. Available: https://jnsecurite2018.sciencesconf.org/data/pages/Jean_Max_DUTERTRE.pdf

[77] B. Selmke, K. Zinnecker, P. Koppermann, K. Miller, J. Heyszl and G. Sigl, "Locked out by Latch-up? An Empirical Study on Laser Fault Injection into Arm Cortex-M Processors," *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Amsterdam, Netherlands, 2018, pp. 7-14, doi: 10.1109/FDTC.2018.00010.

[78] W. Lu, R. Wang, C. Zeng, C. Liu and X. Wang, "A General Fault Injection Method Based on JTAG," *2018 Prognostics and System Health Management Conference (PHM-Chongqing)*, Chongqing, China, 2018, pp. 604-608, doi: 10.1109/PHM-Chongqing.2018.00108.

[80] M. Liu, Z. Zeng, F. Su and J. Cai, "Research on fault injection technology for embedded software based on JTAG interface," *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*, Hangzhou, 2016, pp. 1-6, doi: 10.1109/ICRMS.2016.8050155.

[80] J. Rodriguez, A. Baldomero, V. Montilla and J. Mujal, "LLFI: Lateral Laser Fault Injection Attack," *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Atlanta, GA, USA, 2019, pp. 41-47, doi: 10.1109/FDTC.2019.00014.

[81] "Fault injection", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Fault_injection. [Accessed: 24- Feb- 2021].

[82] "What is fault injection testing? - Definition from WhatIs.com", *SearchSoftwareQuality*, 2021. [Online]. Available: https://searchsoftwarequality.techtarget.com/definition/fault-injection-testing. [Accessed: 24- Feb- 2021].